**NAVAL SURFACE WARFARE CENTER**
**PANAMA CITY DIVISION**
**PANAMA CITY, FL 32407-7001**

TECHNICAL REPORT
NSWC PCD TR-2017-004

# MODULAR ALGORITHM TESTBED SUITE (MATS): A SOFTWARE FRAMEWORK FOR AUTOMATIC TARGET RECOGNITION

Principal Contributors
Derek R. Kolacinski
Advanced Signal Processing and ATR Branch (X23)
Unmanned Systems, Automation, and Processing Division
Science and Technology Department

Bradley C. Marchand
Applied Sensing and Processing Branch (X24)
Unmanned Systems, Automation, and Processing Division
Science and Technology Department

**January 2017**

| REPORT DOCUMENTATION PAGE | *Form Approved OMB No. 0704-0188* |
|---|---|

| | | |
|---|---|---|
| 1. REPORT DATE *(DD-MM-YYYY)* 31-01-2017 | 2. REPORT TYPE Technical | 3. DATES COVERED *(From - To)* |

| | |
|---|---|
| 4. TITLE AND SUBTITLE Modular Algorithm Testbed Suite (MATS): A Software Framework for Automatic Target Recognition | 5a. CONTRACT NUMBER |
| | 5b. GRANT NUMBER |
| | 5c. PROGRAM ELEMENT NUMBER |
| 6. AUTHOR(S) Kolacinski, Derek R. Marchand, Bradley C. | 5d. PROJECT NUMBER DR-041236 |
| | 5e. TASK NUMBER |
| | 5f. WORK UNIT NUMBER |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NAVAL SURFACE WARFARE CENTER, PANAMA CITY DIVISION 110 VERNON AVENUE PANAMA CITY FL 32407-7001 | 8. PERFORMING ORGANIZATION REPORT NUMBER NSWC PCD TR-17/004 |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) OFFICE OF NAVAL RESEARCH ATTN JASON STACK MINE WARFARE & OCEAN ENGINEERING PROGRAMS CODE 32, SUITE 1092 875 N RANDOLPH ST ARLINGTON VA 22203 | 10. SPONSOR/MONITOR'S ACRONYM(S) ONR |
| | 11. SPONSOR/MONITOR'S REPORT NUMBER(S) |

**12. DISTRIBUTION/AVAILABILITY STATEMENT**

(A) Approved for public release: distribution unlimited.

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**

Automatic target recognition (ATR) is a promising technology that could significantly improve naval mine countermeasures (MCM) operations by automating a large portion of the data analysis. Successful long-term implementation of ATR requires a flexible platform to facilitate the development and testing of ATR algorithms. To that end, NSWC PCD has created the Modular Algorithm Testbed Suite (MATS), a modular software architecture for ATR development. The MATS framework provides a common ATR development architecture for disparate development groups and serves as the integration point for introducing ATR into Fleet post-mission analysis (PMA) systems.

**15. SUBJECT TERMS**

Automatic Target Recognition; ATR; Modular Algorithm Testbed Suite; MATS; Mine Countermeasures Operations

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | Derek R. Kolacinski |
| U | U | U | SAR | 24 | 19b. TELEPHONE NUMBER *(Include area code)* (850) 230-7218 |

## FOREWORD

This technical report provides an overview of the Modular Algorithm Testbed Suite (MATS). It describes the architecture, what it is used for, how it is beneficial to the Automatic Target Recognition (ATR) developer community of interest (COI) and why MATS has become the framework of choice for development of ATR at Naval Surface Warfare Center, Panama City Division (NSWC PCD).

This report has been prepared, reviewed, and approved by the Science and Technology Department.

[illegible]

Derek R. Kolacinski
Advanced Signal Processing and ATR Branch (Code X23)

[illegible]

Julia Gazagnaire
Acting Head, Advanced Signal Processing & ATR Branch (Code X23)

[illegible]

Frank J. Crosby
Head, Unmanned Systems, Automation & Processing Division (Code X20)

[illegible]

Kerry W. Commander
Head, Science & Technology Department (Code X)

## CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

## ANNEX

## 1 BACKGROUND

Automatic target recognition (ATR) is a promising technology that could significantly improve naval mine countermeasures (MCM) operations by automating a large portion of the data analysis. Successful long-term implementation of ATR requires a flexible platform to facilitate the development and testing of ATR algorithms. To that end, NSWC PCD has created the Modular Algorithm Testbed Suite (MATS), a modular software architecture for ATR development. The MATS framework provides a common ATR development architecture for disparate development groups and serves as the integration point for introducing ATR into Fleet post-mission analysis (PMA) systems.

## 2 INTRODUCTION

ATR software identifies targets in data measurements via a computer algorithm. In the context of MCM sonar survey operations, this means automatically identifying regions of interest (ROIs) within sonar imagery that are consistent with man-made objects. This is accomplished in three main stages: detection, feature extraction, and classification. Figure 1 illustrates each stage of the ATR process. The detector quickly scans the entire image, identifying statistical anomalies in the image and considerably narrowing the focus of the later stages, which are more computationally-intensive, to a relatively small number of ROIs. During feature extraction, descriptive measurements (i.e., features) are taken at each anomaly location. Finally, these features are used by a pattern classification algorithm, or classifier, to make a decision as to whether each anomaly is consistent with mine-like objects. Decisions are based on a decision boundary that is derived from training data composed of labeled targets and non-targets.

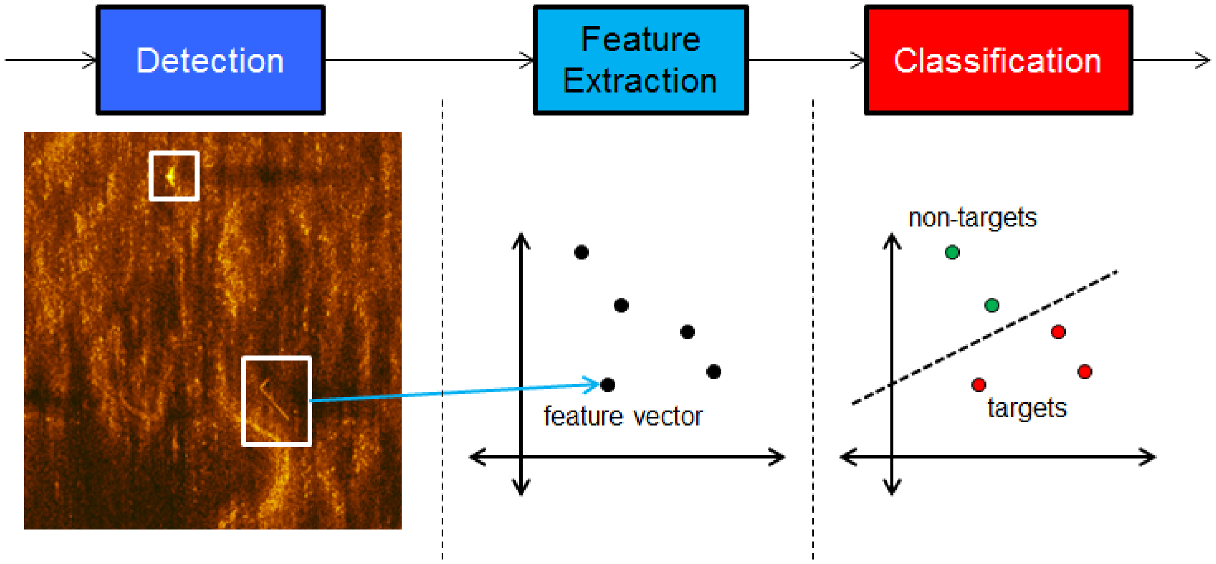


**Figure 1. Main stages of ATR processing**

Due to the multi-stage nature of ATR, different researchers may focus on different aspects of the ATR problem. Ideally, the best approaches would be chosen for each stage to yield a high-performing ATR approach. However, combining algorithms from a wide variety of sources poses a logistical challenge to software integration. Without a coordinated effort to integrate the algorithms into a unified framework, full integration becomes challenging and requires significant effort because a separate interface must be created for each pair of modules implementing adjacent stages in the ATR processing stream. As more algorithms are added, the challenge grows exponentially as each one must be integrated with an increasing number of existing algorithms in adjacent stages. Full integration with all existing components can quickly become overwhelming, even for relatively small numbers of algorithms. This concept is illustrated in Figure 2, where a small set of eight modules have twelve separate interfaces, and a new module (F3) requires an additional six.

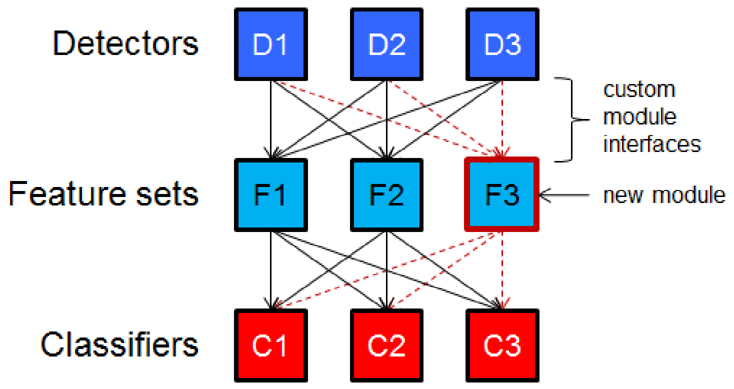


**Figure 2. Non-modular ATR architecture**

A common, modular standard is necessary to alleviate disjointed code development and simplify integration at each stage. Under the modular paradigm shown in Figure 3, a new algorithm will immediately be compatible with all other previously integrated modules as long as it conforms to the common standard. It becomes just another module available for use. This approach significantly lowers the cost and time of algorithm integration and also facilitates incorporating software from Government, academic, and industry partners. The desire for modularity has served as the primary driving force behind the development of the MATS (Modular Algorithm Testbed Suite) framework as a tool for integrating, testing, and evaluating ATR algorithms.

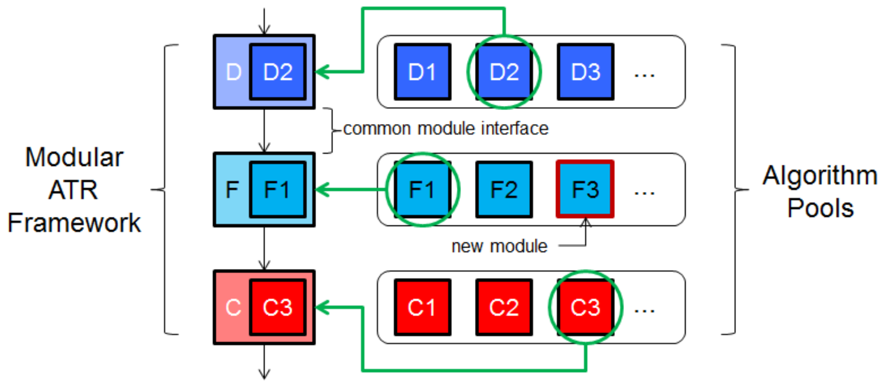


**Figure 3. Modular ATR architecture**

## 3 METHODOLOGY

The MATS architecture can be divided into two broad categories: modules, which perform the bulk of the ATR processing, and the overall framework that connects the modules together. Modules are organized into groups called "module pools," each of which has a specific and unique function to perform within the ATR processing stream. For example:

- A Data Reader converts a data file from some format to the standardized MATS input structure.
- A Detector identifies regions of interest within a given area defined by a set of one or more data files, and returns a list of contacts representing those regions.
- A Feature Generator creates a vector of measurements to represent each contact for the purpose of classification.
- A Classifier discriminates between targets and non-targets using rules derived from past data.

All components within the same pool perform the same task—and are thus completely interchangeable—as long as the defined interface standards are met. Figure 4 is a simplified diagram of the MATS process that illustrates the relationships among these module pools. Each pool is depicted as a colored block that represents a generic algorithm from that pool. Optional components are indicated by blocks with dashed outlines. The white blocks are inherent to the MATS framework and are not changed by the ATR developer.

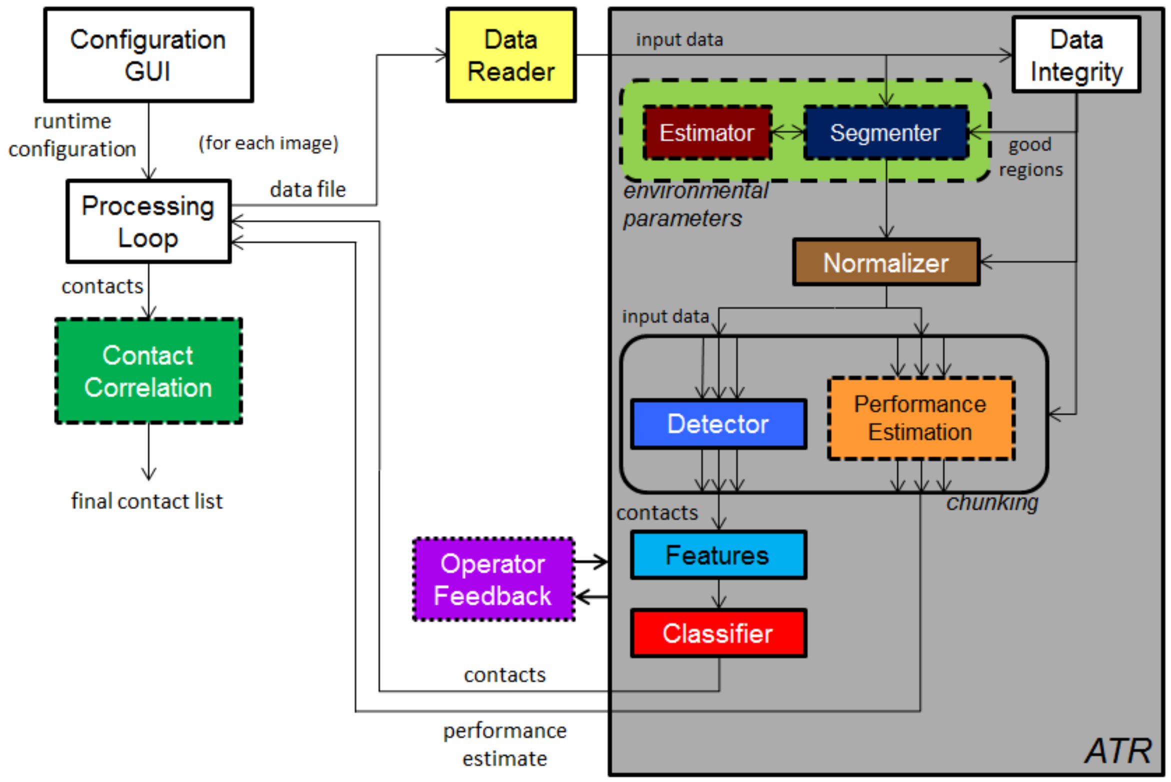


**Figure 4. Diagram of the MATS ATR processing framework**

The MATS framework provides a general communication protocol between the modules using common interfaces and data structures. During the early stages of processing, modules operate on the MATS input structure, which contains sonar imagery over a given area as well as other data related to sonar imagery collection, such as image resolutions and navigation data. MATS was created for side-look sonar imagery, but any similar 2D side-looking sensor can be accommodated fairly easily. Upon completion of the Detector, modules operate on a list of contacts, each of which represents an image location that has been identified by the Detector as a potential target. A contact encapsulates all relevant data at that location, including coordinates, image features, and image snippets. The contact list eventually becomes the primary output of the entire ATR process.

## 3.1 MATS ATR Processing Walkthrough

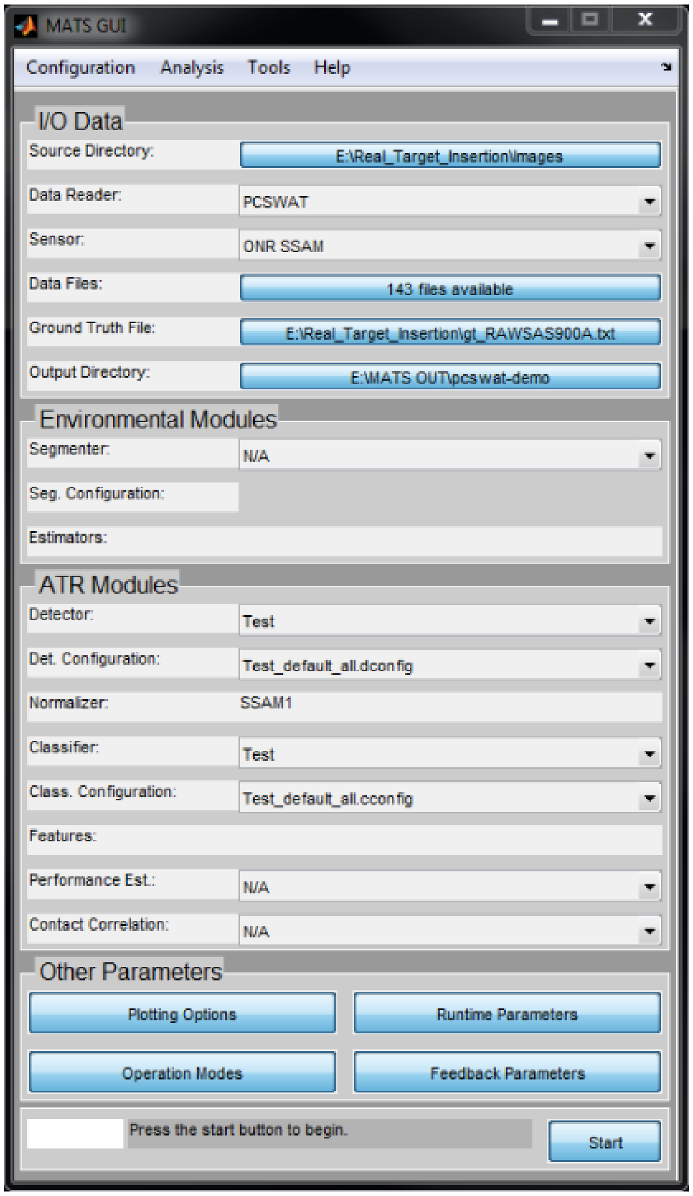
The following section is a walkthrough for a simple MATS ATR process that covers the entire MATS processing chain, from the initial configuration, through the ATR modules, to a final list of contacts. The more advanced capabilities of MATS are described briefly at the end of the walkthrough. More information can be found in the MATS User Manual (Annex A), which covers all functions of the MATS software package in more detail.

The first step in processing ATR via MATS is configuring the runtime parameters using the MATS GUI. MATS runtime parameters fall into three general categories. The first group defines what input data are available, how these inputs are organized, and where the ATR outputs will be saved. The second group specifies which ATR modules will be executed during processing through a series of drop-down menus. The last group contains several buttons leading to various menus for additional optional parameters; these are only used in advanced operating modes and can be ignored for this example.

Before the ATR can begin, valid module configuration files must be selected for all configurable modules. In the simplest case, this refers to both the Detector and the Classifier. Configuration files are produced either manually or by a training GUI with an automated training procedure. Automated training requires an algorithm-specific training function that takes in labelled data and returns a structure containing all of the necessary parameters required to successfully run the module. This structure is included in a configuration file and will be included as an argument whenever the module is invoked with that configuration file at runtime. The list of configuration files available for a particular module is filtered based on runtime parameters such as sensor type so that only configurations compatible with the current settings can be selected.

The sample parameters that will be used throughout this example are shown in Figure 5. The selected data directory has 143 pairs of sonar images. This data will be processed using the Test Suite of ATR modules, a minimal set of modules used for development and debugging that is included in the MATS package. A ground truth file, which is a text file that lists the correct locations of actual targets, is also included so that performance analyses may be performed on the results after processing.

Once ATR processing begins, the contents of the specified input directory are analyzed and grouped into complete sets of one or more co-located images. What constitutes a complete set depends on the requirements of the selected sensor. In this example, data is organized in pairs of high- and low-frequency imagery, as shown in Figure 6. The imagery contained in one such pairing is shown in Figure 7 and Figure 8. Both data files must be present to make a complete set; the remaining unmatched files, colored gray in Figure 6, will be ignored. The ATR processing loop iterates over the list of complete sets found within the input directory, passing each set of images into the ATR routine individually and resulting in a list of contacts identified in those images.



Source Directory: E:\Real_Target_Insertion\images
Data Reader: PCSWAT
Sensor: ONR SSAM
Data Files: 143 files available
Ground Truth File: E:\Real_Target_Insertion\gt_RAWSAS900A.txt
Output Directory: E:\MATS OUT\pcswat-demo
Segmenter: N/A
Seg. Configuration:
Estimators:
Detector: Test
Det. Configuration: Test_default_all.dconfig
Normalizer: SSAM1
Classifier: Test
Class. Configuration: Test_default_all.cconfig
Features:
Performance Est.: N/A
Contact Correlation: N/A

Press the start button to begin.

**Figure 5. MATS Configuration GUI**

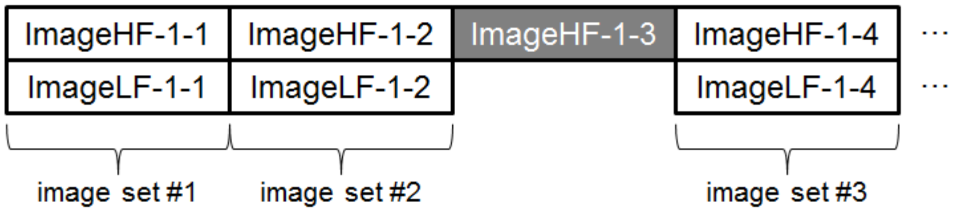


**Figure 6. Complete sets of matching images for a dual-frequency sensor**

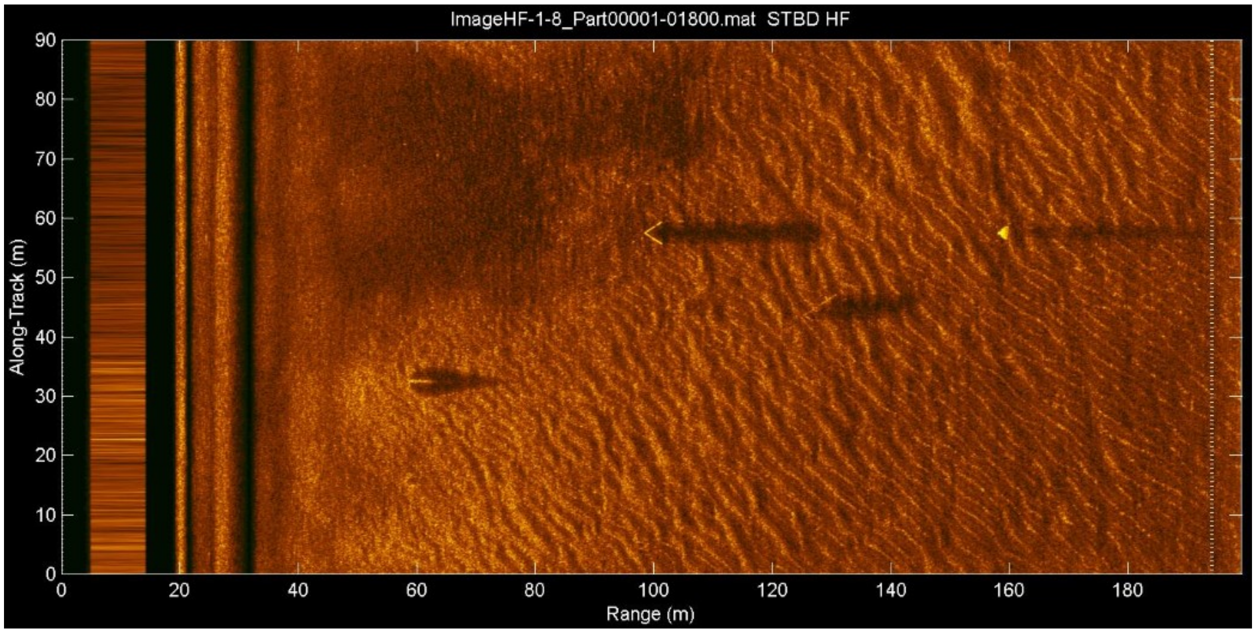


**Figure 7. Sample high-frequency imagery for example data set**

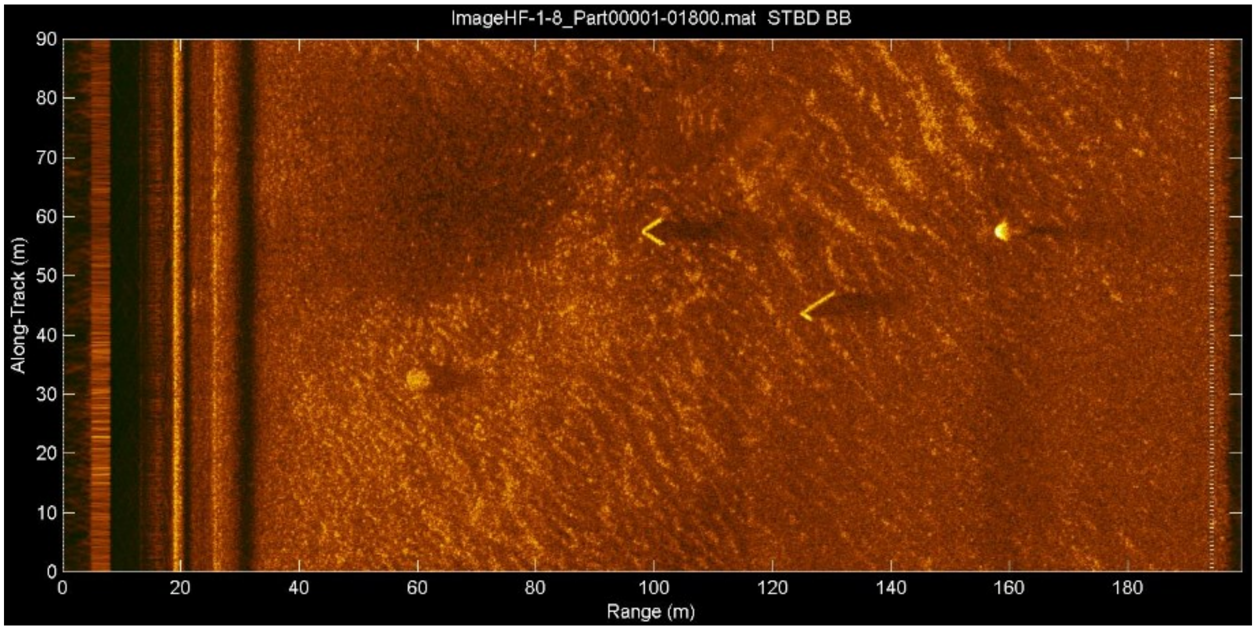


**Figure 8. Sample low-frequency imagery from example data set**

For some data formats, a single set of input data files contains more data than is practical to process all at once. In this case, MATS partitions a complete set of input data into a series of overlapping sections called "image chunks." Each image chunk is roughly square and avoids portions of the input data that fail built-in integrity checks. For example, vehicle turns distort the imagery, so MATS attempts to detect all significant turns and omits these sections of data from processing. The result resembles the diagram in Figure 9, where the green regions of the data have passed the integrity checks and red regions have failed them. Each passing region is normalized separately and then chunked. The chunks are processed sequentially through the Detector and Performance Estimation modules and the results are combined as if all the input data was processed as a whole.

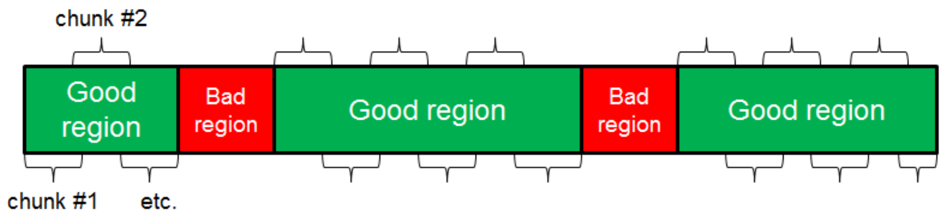


**Figure 9. Example of input data requiring chunking**

The data reader module reads the files in a single complete set of input data and returns a standard MATS input structure. The resulting MATS input structure must conform to the specifications in Table 1 so that later modules can properly access the input data. Fields with a "bb" prefix are only used for multi-frequency data formats. Additional information regarding this and other data structures may be found in Annex A.

**Table 1. Selected fields of the MATS input structure**

| Field | Data Type | Description |
|---|---|---|
| .hf | matrix (float) | High frequency (HF) image (complex) |
| .hf_cres | float | Cross-track (range) resolution of the HF image (in m) |
| .hf_ares | float | Along-track resolution of the HF image (in m) |
| .hf_cnum | integer | Cross-track (range) dimension of the HF image (in pixels) |
| .hf_anum | integer | Along-track dimension of the HF image (in pixels) |
| .bb | matrix (float) | Broadband (BB) image (complex) |
| .bb_cres | float | Cross-track (range) resolution of the BB image (in m) |
| .bb_ares | float | Along-track resolution of the BB image (in m) |
| .bb_cnum | integer | Cross-track (range) dimension of the BB image (in pixels) |
| .bb_anum | integer | Along-track dimension of the BB image (in pixels) |
| .side | string | Side of vehicle from which the image was taken = {'PORT', 'STBD'} |
| .lat | array (float) | Latitude of vehicle (in degrees) |
| .long | array (float) | Longitude of vehicle (in degrees) |
| .fn | string | Filename of image file/some other image ID |
| .havegt | integer | 1 = ground truth is available; 0 = no GT available |
| .gtimage | structure | Ground truth data for this image |
| .heading | array (float) | Nominal heading of the vehicle (in radians) |
| .time | array (float) | Time stamp |
| .sensor | string | Sensor ID string |
| .mode | character | Run-time mode: 'A' = ATR only; 'P' = performance estimation only; 'B' = both |
| .sweetspot | array (integer) | Range pixel bounds on area of image suitable for ATR |

The Test Detector used in this example creates contacts at all available ground truth locations in the current file and adds a small number of contacts in random locations. In other words, it has perfect detection with a few random false alarms. It is not a "real" detector in that the results are completely dependent on the availability of accurate ground truth data; however, it is still an instructive example of how MATS works. Table 2 and Table 3 describe some of the fields of the output data structures that are generated by a detector. Contact data is divided between the contact structure, which contains the most commonly used contact data, and the extra contact data structure, which contains fields that are either rarely used or significant in size and are therefore saved to a supplementary file. Thus basic information about a contact like its pixel coordinates are contained in the contact structure, while contact snippets are saved in the supplementary file.

**Table 2. Selected fields of the MATS contact structure**

| Field | Data Type | Description |
|---|---|---|
| .uuid | string | Unique identifier for each contact |
| .x, .y | integers | Pixel location of contact in the image (centroid) |
| .features | array (float) | Features used to classify contact |
| .fn | string | Filename of image file (or other image ID) |
| .side | string | Side of vehicle: {'PORT', 'STBD'} |
| .gt | integer | Ground truth value of contact (if known) |
| .class | integer | Classification of contact (1 = target; 0 = non-target) |
| .classconf | float | Probability that contact is target |
| .detector | string | Detector module ID |
| .featureset | string | Feature set module ID |
| .classifier | string | Classifier module ID |
| .ecdata_fn | string | Filename containing remaining contact data |

**Table 3. Selected fields of the extra contact data structure**

| Field | Data Type | Description |
|---|---|---|
| .uuid | string | Unique identifier for each contact |
| .hfsnippet | matrix (float) | Area around contact in HF image (complex) |
| .bbsnippet | matrix (float) | Area around contact in BB image (complex) |
| .lat | float | Latitude of contact (in degrees) |
| .long | float | Longitude of contact (in degrees) |
| .heading | float | Nominal heading of the vehicle (in radians) |
| .time | float | Timestamp (UTC) |
| .hf_cres | float | HF image cross-track resolution (in m) |
| .hf_ares | float | HF image along-track resolution (in m) |
| .hf_cnum | integer | HF image cross-track dimension (in pixels) |
| .hf_anum | integer | HF image along-track dimension (in pixels) |
| .bb_cres | float | BB image cross-track resolution (in m) |
| .bb_ares | float | BB image along-track resolution (in m) |
| .bb_cnum | integer | BB image cross-track dimension (in pixels) |
| .bb_anum | integer | BB image along-track dimension (in pixels) |

The Feature modules generate a numerical representation of the contact by operating on one or more image snippet(s) like the one shown in Figure 10. Each snippet contains the contact at the center surrounded by the immediate background. For multi-frequency data, both snippets cover the same area, although the pixel sizes of the snippet may differ if the images have different resolutions. The specific Feature modules executed depends on the contents of the classifier configuration file selected during the initial configuration step. In this example, which is intended to be a starting point, no contact features are calculated.

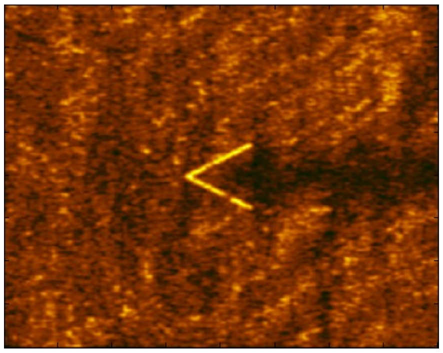


**Figure 10. Image snippet for a sample contact**

The Test Classifier used in this example classifies contacts according to the available ground truth data; each contact is classified as a target only if a ground truth object is nearby and is assigned a random confidence value between zero and one. ATR results for each file are summarized in the MATS display as colored highlight boxes over the original imagery as depicted in Figure 11. Ground truth locations are marked by a blue box; red and green boxes represent contact locations that are classified as targets and non-targets, respectively. A copy of this display window is saved as a JPEG image file in the output directory, and can be used as a simple, visual indicator of ATR performance.

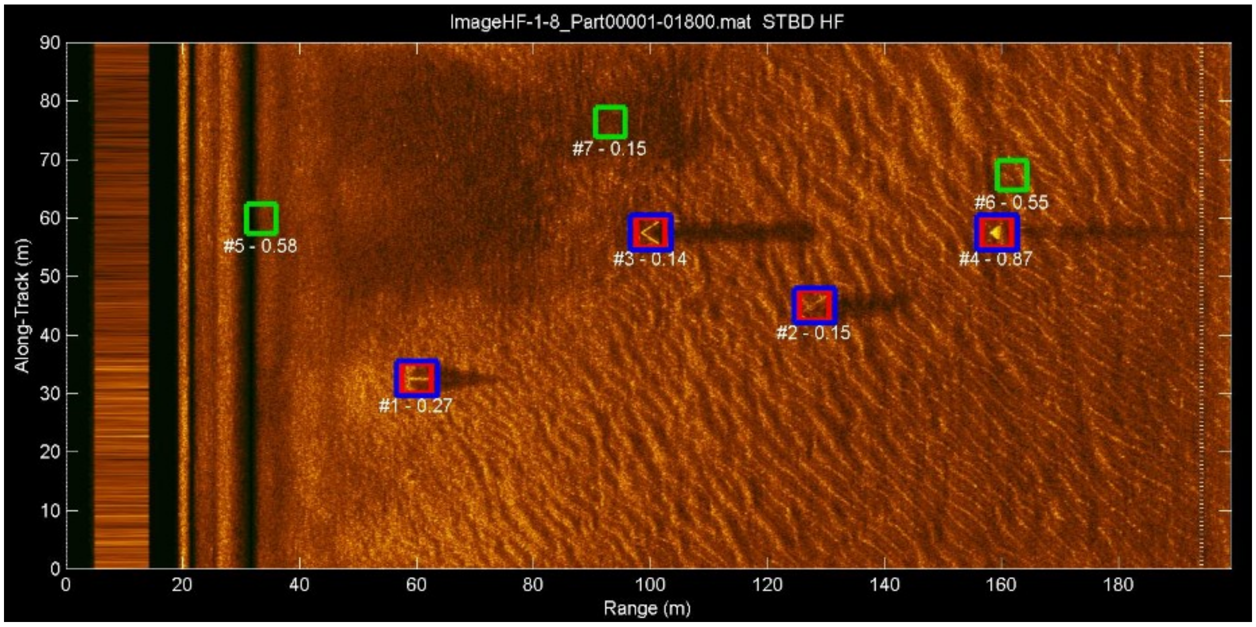
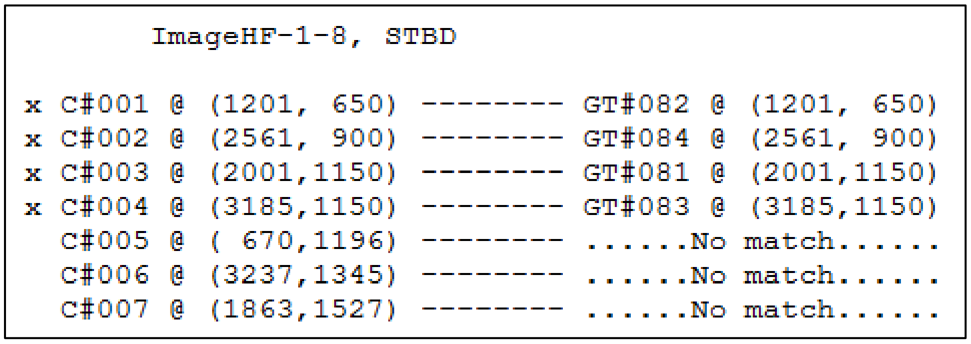


**Figure 11. Sample output image with contact and ground truth highlights**

Upon completing the ATR processing on the last set of images, MATS creates a data structure that compares the resulting contacts with the objects listed in the ground truth file. The data structure is an array such that each array element contains a list of all contacts and ground truth entries, matched if possible, that were generated by processing the corresponding set of images. It functions as a concise summary of which objects the ATR found and which ones it missed. MATS includes several analysis scripts that can parse and interpret this output data structure. One such script prints the comparison data structure directly. The excerpt associated with the sample image set is shown in Figure 12. This example contains four contacts that match nearby ground truth objects followed by three additional false detections. The first four contacts have also been classified as targets and are thus denoted by an initial

'X.' These results are consistent with the expected behavior of the Test Detector and Test Classifier modules.

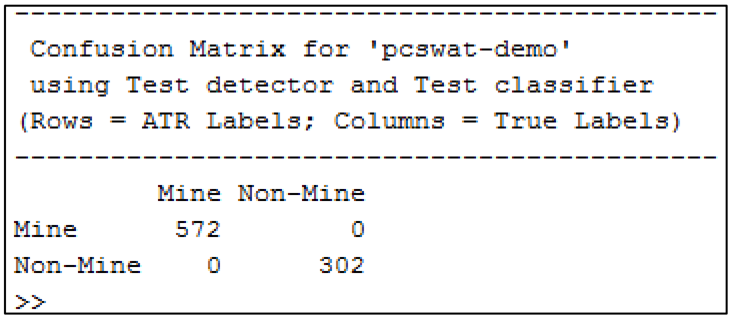


```
        ImageHF-1-8, STBD

x C#001 @ (1201, 650) -------- GT#082 @ (1201, 650)
x C#002 @ (2561, 900) -------- GT#084 @ (2561, 900)
x C#003 @ (2001,1150) -------- GT#081 @ (2001,1150)
x C#004 @ (3185,1150) -------- GT#083 @ (3185,1150)
  C#005 @ ( 670,1196) -------- ......No match......
  C#006 @ (3237,1345) -------- ......No match......
  C#007 @ (1863,1527) -------- ......No match......
```

**Figure 12. Contact vs. ground truth comparison for sample image**

Several automated MATS tools use the comparison data structure to create data products such as confusion matrices and Receiver Operator Characteristic (ROC) curves for analyzing ATR performance. For this scenario, the confusion matrix shown in Figure 13 has no errors because the classification of each contact was derived from ground truth. The ROC curve depicted in Figure 14 is almost a worst-case straight line, which is to be expected given that the confidence values on which the ROC curve is based were assigned randomly. Other tools can sort and display contact field data, or display the execution time of each module on each set of input data files.



```
--------------------------------------------
 Confusion Matrix for 'pcswat-demo'
 using Test detector and Test classifier
(Rows = ATR Labels; Columns = True Labels)
--------------------------------------------
         Mine Non-Mine
Mine       572        0
Non-Mine     0      302
>>
```

**Figure 13. Confusion matrix for MATS walkthrough scenario**

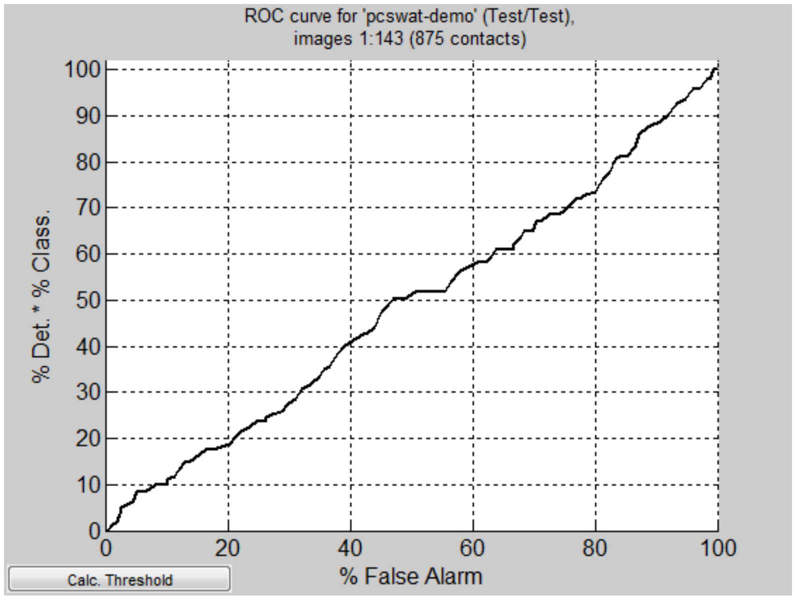
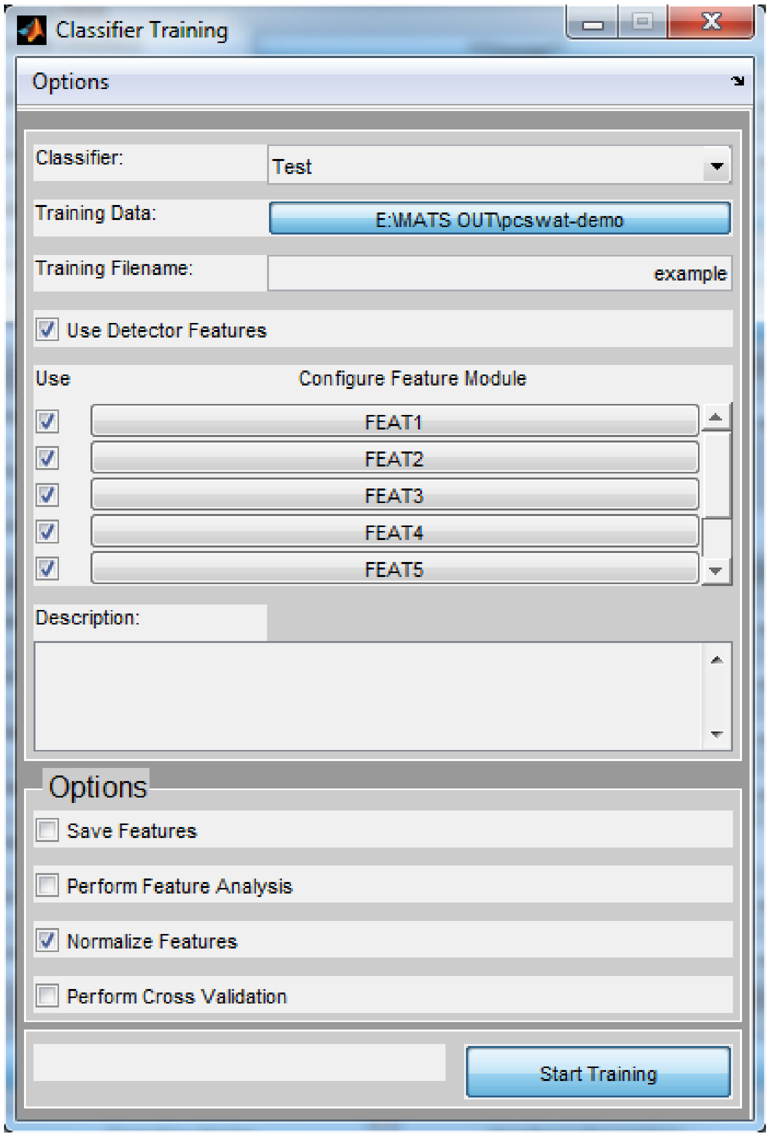


**Figure 14. ROC curve for MATS walkthrough scenario**

The contacts created in this simple example may be used in the future to create a more capable Classifier module configuration. The Classifier Training GUI, shown in Figure 15. MATS Classifier Training GUIFigure 15, can train a selected Classifier module on any set of compatible Feature modules using an existing set of contacts as training data. Each Feature module can be configured further to suit its needs by clicking on its respective button in the list. During training, the desired features are calculated for each contact and normalized. These features are used by the chosen Classifier module's training function to calculate the necessary parameters for that module. If sufficient data is available, cross-validation is available to generate a more robust classifier configuration. The final result is a new classifier configuration file containing the newly calculated parameters that may be selected through the MATS Configuration GUI for future ATR processing.



Classifier Training

Options

Classifier: Test

Training Data: E:\MATS OUT\pcswat-demo

Training Filename: example

☑ Use Detector Features

| Use | Configure Feature Module |
|---|---|
| ☑ | FEAT1 |
| ☑ | FEAT2 |
| ☑ | FEAT3 |
| ☑ | FEAT4 |
| ☑ | FEAT5 |

Description:

Options

☐ Save Features

☐ Perform Feature Analysis

☑ Normalize Features

☐ Perform Cross Validation

Start Training

**Figure 15. MATS Classifier Training GUI**

An optional feature analysis will be performed if the appropriate checkbox is enabled in the training GUI. This generates a plot of the likely significance of each feature to classification. An example is shown in Figure 16. Features calculated within the same Feature module have the same color; in this case, four Feature modules were used. In addition, the most significant features are indicated by their ranking. This information can be used to prune the feature list to include only the most meaningful features.

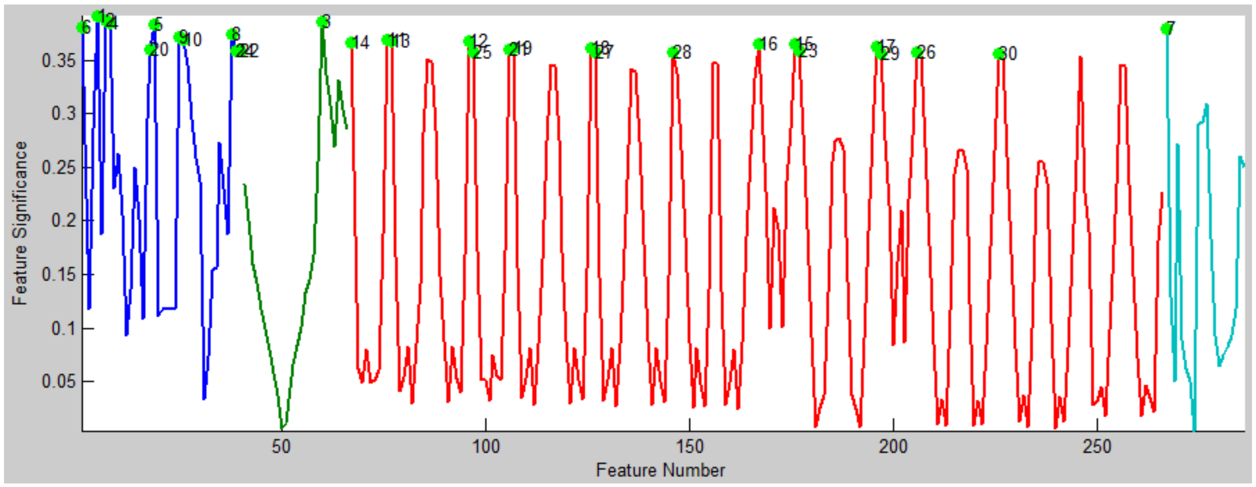


**Figure 16. Feature analysis of four Feature modules**

## 3.2 Advanced Use Cases

In addition to the essential modules just described, there are several optional module types that can be enabled within MATS if desired. The environmental parameter modules, Image Segmenters and Parameter Estimators, add a segmentation map and some number of descriptive values to the MATS input structure. The ATR processing stream can then use this information to become adaptive. For example, a module might use a higher threshold in a rocky area than one with a smooth bottom, or perhaps a different module is used altogether.

MATS also allows a Performance Estimation module to be run in parallel to the standard ATR processing stream in order to provide an assessment of expected performance of ATR on the available data. While performance estimation is not required for ATR execution, it can be useful for establishing trust with the user, which is critical for enabling more sophisticated ATR use cases such as a fully autonomous system.

If several instances of MATS exist on the same network, they may take advantage of the MATS distributed mode to process large quantities of data more quickly. In this mode, all MATS instances are part of a common group that works collaboratively on common ATR tasks. One instance is designated the master process and manages the ATR tasks assigned to the group, which includes creating work packages for the other instances to complete. In this manner, ATR may be performed on several data files simultaneously, resulting in significantly faster progress through an ATR task.

# 4 DISCUSSION

From its inception in 2010, MATS has matured into the standard framework for ATR development at NSWC PCD. One major reason for this is that the modular architecture allows algorithm developers to focus on a particular module type while using existing modules for the rest of the ATR process. This is especially useful when developing modules for the later stages of ATR. For example, a classifier must eventually be trained on labeled contact data. Instead of creating a separate mechanism to generate contact data with features, the contact data can be generated easily using MATS itself, saving the developer time and effort.

Another benefit of the modular architecture is that it is convenient for integrating algorithms produced by outside organizations. Only the common interfaces of a module are relevant to MATS integration, so it is not necessary for module developers to have detailed knowledge of the inner workings of the MATS framework. Conversely, the internal details of individual MATS modules need not be revealed to the greater MATS community. This is particularly relevant for contractors who may want to protect intellectual property.

MATS also provides ample opportunity for future development. MATS already operates on a specified configuration, so it can easily be extended into an adaptive system by creating logic that determines which one of several existing configurations should be used. If specific conditions that cause an existing module to fail are encountered in the future, a new module—possibly an entirely different approach—may be developed to address the deficiency. The new module can be invoked thereafter whenever the problematic conditions are identified, resulting in a more flexible and more robust ATR process.

The MATS ATR framework has become a platform for transitioning ATR software to the Fleet because of these advantages. The modular architecture allows the best modules from a variety of sources to be included in the final product. While most of the deployed code has been converted from MATLAB to C for improved performance, new MATLAB-based ATR modules can still be integrated into the C-based ATR through the original MATS processing stream, which compiles the MATLAB ATR code into a C library. In this way, a new algorithm can rapidly transition from an idea to part of a deployed system.

## 5 CONCLUSIONS

MATS incorporates several important features that have streamlined ATR development and testing at NSWC PCD. As a development framework, the MATS common architecture and modular design promote interoperability and simplify the integration of both current and future research in a variety of topics related to ATR. As a testing suite, MATS can process large amounts of data and analyze the results in an automated fashion. As an integration platform, MATS can draw upon a diverse research community for the most suitable ATR algorithms. For these reasons, MATS has the potential to enhance naval MCM capabilities in the future.

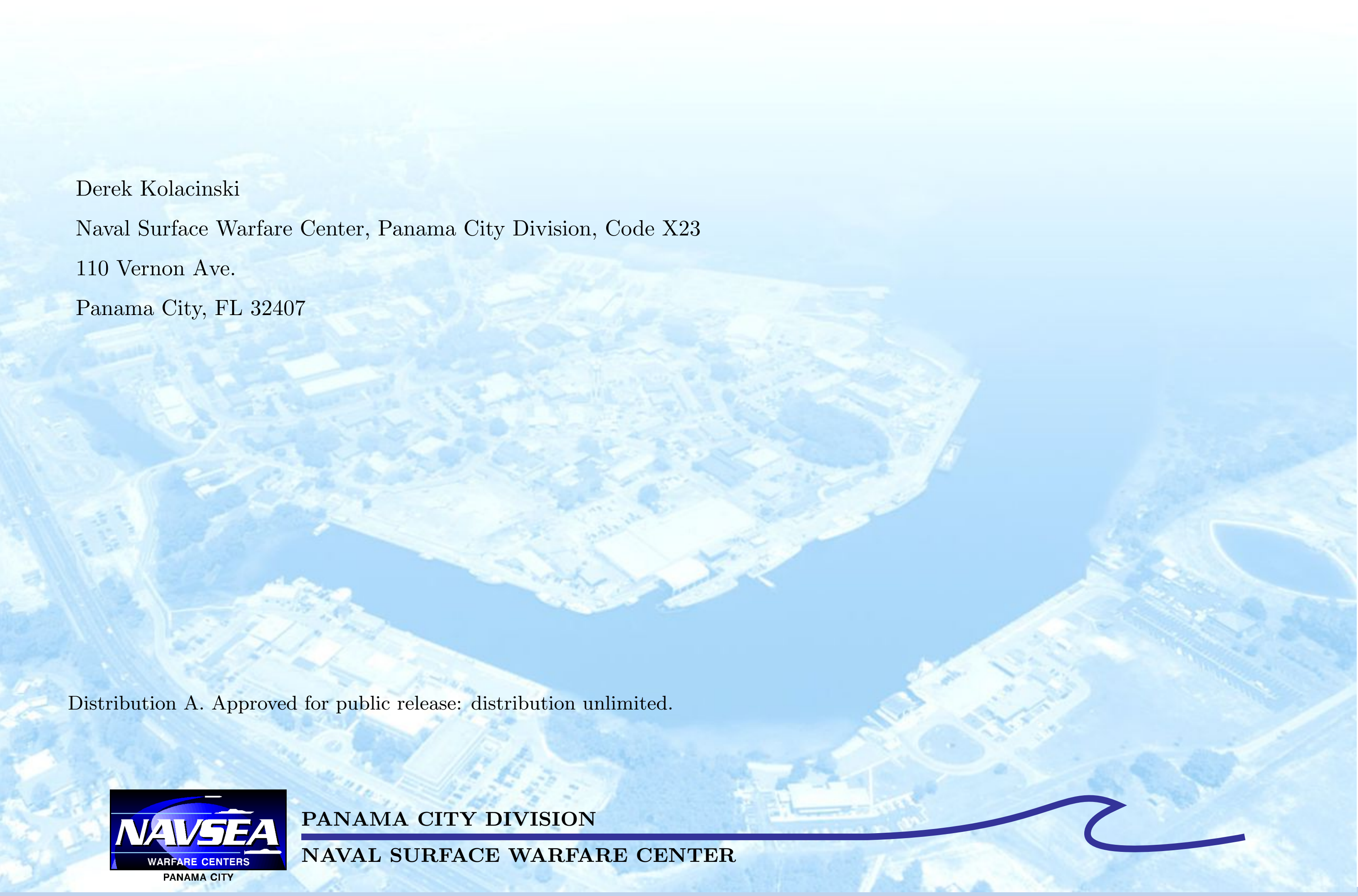
# Annex A - MATS USER MANUAL, VERSION 3.2

Naval Sea Systems Command

**Naval Surface Warfare Center**
**Panama City**

# Modular Algorithm Testbed Suite (MATS)

# User Manual



Derek Kolacinski

Naval Surface Warfare Center, Panama City Division, Code X23

110 Vernon Ave.

Panama City, FL 32407

PANAMA CITY DIVISION

NAVAL SURFACE WARFARE CENTER

# Contents

# List of Figures

# List of Tables

# Introduction

The Modular Algorithm Testbed Suite (MATS) is a software package for developing and testing automatic target recognition (ATR) algorithms. It employs an open, modular architecture to provide developers with a simple way to test and compare ATR algorithms quickly. MATS is written using the MATLAB® programming language and requires both MATLAB® version 2009b or later and the Image Processing Toolbox to run properly. In addition, certain analysis tools require the Statistics Toolbox as well.

To start MATS, launch MATLAB® and set the current working directory to the root directory of the MATS package. This can be done by using either the MATLAB® command prompt (indicated by `>>`) or the 'Current Folder' panel in the graphical user interface (GUI). Then, type `MATS_GUI` into the command prompt to run the file `MATS_GUI.m`. This creates the GUI for configuring a variety of MATS run-time parameters. After the necessary parameters are set, pressing the 'Start' button located at the bottom of the configuration GUI will begin the ATR processing.

## Document Overview

This document is intended to provide a thorough overview of the various aspects of the MATS package. Chapter 1 (Basic Operation) provides the necessary information for basic users. Sections 1.1 (MATS Modules) and 1.2 (Data Structures) describe the basic architecture of MATS. Section 1.3 (Configuring the GUI) details the GUI configuration options and Section 1.4 (Running MATS) describes expected run-time behavior. The remaining subsections cover some more sophisticated use cases for advanced users.

Chapter 2 (Developer Reference) includes several sections that explore some of the underlying mechanics of MATS. Procedures for integrating custom modules are included in Section 2.2 (Adding a Custom Module to MATS). Additional information regarding feedback classifier integration is included in Section 2.3 (ATR and Operator Modules). Sections 2.4 (Supporting Data Structures), 2.5 (Notable Functions), and 2.6 (Data Files) highlight a selection of important data structures, functions, and files, respectively.

## Distribution

This document, the source code of the MATS framework, and the included sample modules (`Test`, `Test_multi`, and `No_extra`), are all Distribution A and may be freely distributed. However, this may ***not*** be true of all modules located within the MATS directory structure. **If you have received more restricted algorithms, appropriate measures must be taken to avoid distribution of these algorithms to unauthorized entities.**

# Chapter 1

# Basic Operation

## 1.1 MATS Modules

The bulk of the processing occurs in the individual components of the MATS system, which are called *modules*. Modules are organized into various categories, or *pools*, based on their function. Each pool has a specific and unique function to perform within the ATR processing stream:

- A ***data reader*** converts a data file from some format to the standardized MATS input structure.
- A ***detector*** takes an image and returns a list of contacts representing objects of interest in the area covered by the provided image(s).
- A ***feature generator*** creates a feature vector to represent each contact for the purpose of classification.
- A ***classifier*** discriminates between targets and non-targets.
- A ***performance estimation*** algorithm estimates an expected level of ATR performance based on properties of the provided image(s).
- A ***contact correlation*** algorithm identifies which contacts are actually the same object from different images.

All components within the same pool perform similar tasks—and are thus completely interchangeable—as long as the data standards defined in Section 1.2 (Data Structures) are met. Figure 1.1 is a simplified diagram of the MATS process that illustrates the relationships among these module pools. Each pool is depicted as a colored block that represents a generic algorithm from that pool. The user chooses which specific algorithms will be executed at run time via the MATS configuration GUI's module selection menus, which are detailed in Section 1.3.3 (ATR Modules).

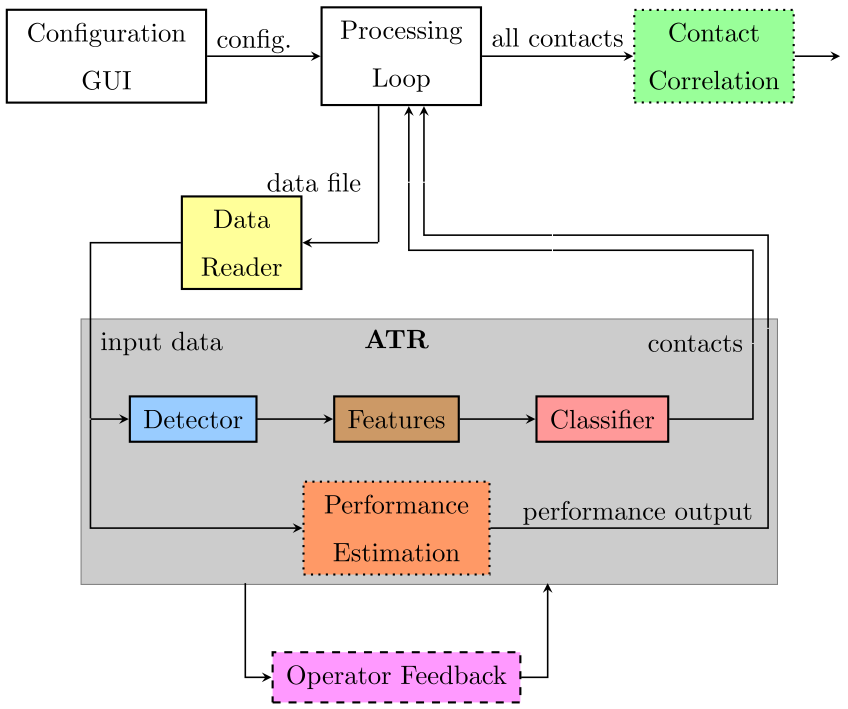


Figure 1.1: MATS Flow Diagram

### 1.1.1 Environmental Modules

The performance of the ATR modules may be improved by the addition of optional *environmental modules*, which extract environmental measurements from the available imagery, to the processing stream. The environmental measurements will be passed on to the ATR modules to allow for more nuanced behavior. For example, a detector may use different threshold values for different types of backgrounds.

The environmental modules contain estimators, which return measurements over a given region, and segmenters, which separate the image into regions. These two modules interact to create a segmentation map and set of environmental measurements that downstream ATR modules can use to adapt to the local environment.

### 1.1.2 Data Integrity Modules

In addition to the modules described in the previous section, there are several components of the MATS architecture that, while not configurable, are still noteworthy. Prior to the execution of the detector, the input data is analyzed by several data integrity modules that attempt to locate regions with problematic data so that they can be removed from the processing stream. The remaining data is then separated into smaller chunks, each of which will be processed by the detector and performance estimation algorithms separately; the anomalous data is ignored. Figure 1.2 illustrates this process. In this example, there are three chunks

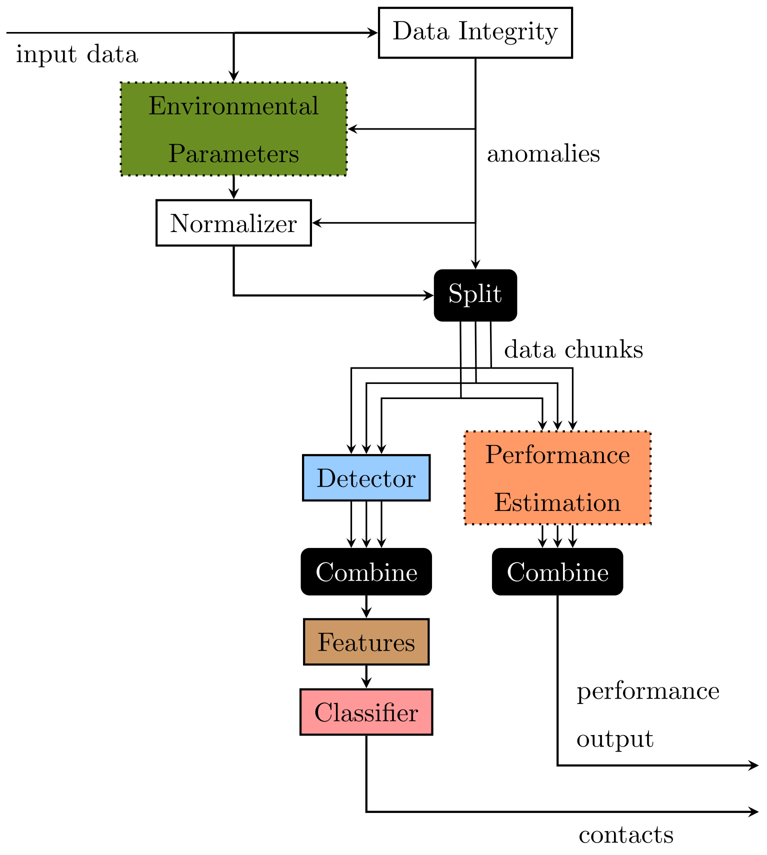
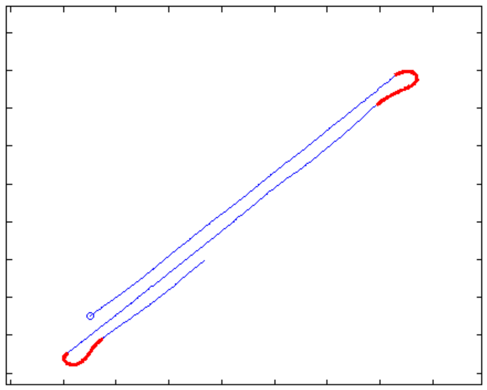


Figure 1.2: Closeup of anomaly detection and data chunking mechanisms

for one image. Depending on the size of the image, there can in fact be many more than three chunks per image.

Currently, there are two data integrity modules. The first detects vehicle turns from the vehicle's navigation data. ATR is disabled during turns because the image is usually significantly distorted and spurious false alarms may occur as a result. The second detects height anomalies, which are defined as unrealistic changes in the vehicle's height data. These can introduce significant errors to calculations for the effective operating range of the ATR, meaning that either a portion of the image data will be ignored, or that regions at the extreme near and far ranges will be included, potentially causing false alarms as well. The combination of these two algorithms filters out common data aberrations from the sonar data, leaving only the straight segments for ATR processing.

Figure 1.3 shows examples of these two phenomena. In Figure 1.3a, a sample vehicle track is shown with the turns highlighted in red. The remaining straight, blue sections are processed by the ATR algorithms. Figure 1.3b shows sample vehicle height data in black. The accompanying blue curves form a binary mask to indicate which regions are considered anomalous. In this example, there are two such regions: a sudden jump and a much gentler oscillation. The jump is clearly impossible and is the result of bad data. The oscillation is


(a) Sample vehicle track with turns

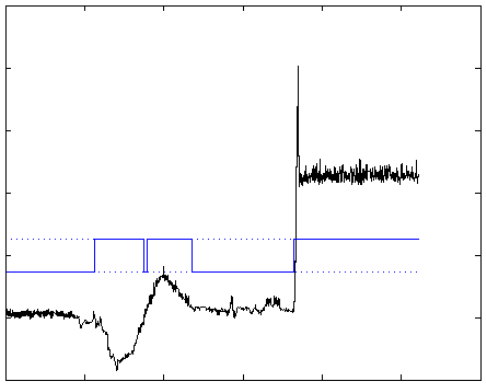

(b) Sample height anomaly

Figure 1.3: Examples of omitted regions

much more plausible and is probably the result of a rolling motion. In each case, the accompanying imagery is distorted to the point of being unusable.

## 1.2 Data Structures

The data structures described in this section are the means by which the different modules communicate with each other. The standardization here plays a crucial role in making the MATS suite a powerful, modular tool.

### 1.2.1 Input Structure

The input structure contains all of the data that is passed into the ATR module. It represents an image and other data that is associated with an image (e.g., image resolutions, vehicle data, etc.). The complete list of input structure fields is listed in Table 1.1.

Table 1.1: Fields of the Input Structure

| **Field** | **Data Type** | **Description** |
|---|---|---|
| .hf | matrix (float) | High frequency (HF) image (complex) |
| .hf_cres | float | Cross-track (range) resolution of the HF image (in $m$) |
| .hf_ares | float | Along-track resolution of the HF image (in $m$) |
| .hf_cnum | int | Cross-track (range) dimension of the HF image (in pixels) |
| .hf_anum | int | Along-track dimension of the HF image (in pixels) |
| .bb | matrix (float) | Broadband (BB) image (complex) |

(Continued)

| **Field** | **Data Type** | **Description** |
|---|---|---|
| .bb_cres | float | Cross-track (range) resolution of the BB image (in $m$) |
| .bb_ares | float | Along-track resolution of the BB image (in $m$) |
| .bb_cnum | int | Cross-track (range) resolution of the BB image (in pixels) |
| .bb_anum | int | Along-track resolution of the BB image (in pixels) |
| .side | string | side of vehicle from which the image was taken = {'PORT', 'STBD'} |
| .lat | array (float) | latitude of vehicle (in degrees) |
| .long | array (float) | longitude of vehicle (in degrees) |
| .fn | string | filename of image file/some other image ID |
| .havegt | int | 1 = ground truth data available; 0 = no ground truth available |
| .gtimage | array (struct) | ground truth info for this image *(See Table 1.2)* |
| .targettype | string | type of target the detector is trying to detect[1] |
| .perfparams | struct | parameters required for the performance estimation code *(See Table 1.3)* |
| .heading | array (float) | nominal heading of the vehicle (in radians) |
| .time | array (float) | time stamp (UTC) |
| .sensor | string | sensor ID string |
| .mode | char | run-time mode: 'A' = ATR only; 'P' = performance estimation only; 'B' = both |
| .sweetspot | array (int) | range pixel bounds on area of image suitable for ATR |
| .environ_params | struct | environmental parameters extracted from the image(s) |

Table 1.2: Fields of the .gtimage Substructure

| **Field** | **Data Type** | **Description** |
|---|---|---|
| .x, .y | arrays (int) | location of ground truth object (in pixels) |
| .side | string | side of vehicle where the ground truth object is located: 'PORT', 'STBD' |
| .num | array (int) | identifier for ground truth object |

(Continued)

[1]Valid options include: 'wedge', 'truncated cone', 'cylindrical', 'torpedo', 'sphere', 'bomb', 'box', 'can', 'coffin', 'disk', 'ellipsoidal', 'elongated sphere', 'fish', 'hemisphere', 'irregular', 'ovoid', 'rectangular box', 'sphere with horns', 'teardrop', 'triangular', 'unspecified'

| Field | Data Type | Description |
|---|---|---|
| .fn | string | filename of image file/some other image ID in which this ground truth object is located |
| .type | int | object type (for multi-class classifiers only): -99 = unknown |

Table 1.3: Fields of the .perfparams Substructure

| Field | Data Type | Description |
|---|---|---|
| .height | array (float) | height of the vehicle at each ping (in $m$) |
| .depth | array (float) | depth of the vehicle at each ping (in $m$) |
| .maxrange | float | maximum range (in $m$) |

Table 1.4: Fields of the .environ_params Substructure

| Field | Data Type | Description |
|---|---|---|
| .map | array (float) | $RxC$ segmentation map with $N$ regions |
| .values | array (float) | $NxV$ array of estimator values |
| .segmenter | string | segmenter algorithm ID string |
| .estimators | cell (string) | list of estimator algorithm ID strings |

### 1.2.2 Performance Estimation

The performance estimation structure provides several metrics that can be used to estimate how the ATR is likely to perform on the provided image(s). This structure is only returned when a performance estimation algorithm is used during the ATR processing.

Table 1.5: Fields of the Performance Estimation Output Structure

| Field | Data Type | Description |
|---|---|---|
| .pdpc | matrix (float) | estimated probability of detection and classification at each pixel (same size as input image) |
| .lfa | matrix (float) | estimated likelihood of false alarm at pixel |
| .fapi | float | estimated number of false alarms per image |
| .B | float | average estimated $P_d P_c$ (whole image) |

(Continued)

| Field | Data Type | Description |
|---|---|---|
| .ATRstatus | char | stoplight performance indicator: • 'r' = red ($P_d P_c < .5$ OR $LFA > .5$ OR number of positive contacts in current image > 10) • 'y' = yellow ($P_d P_c < .7$ OR $LFA > .3$ OR number of positive contacts in current image > 5) • 'g' = green (otherwise) |

### 1.2.3 Contact List

The contact list is an array that represents the main output of the the ATR processing. Each element of the array represents a potential object in the input imagery that was detected by the ATR. The fields associated with it are listed in Table 1.6 . The background color of the field entry indicates when the actual field values are generated using the same color scheme established in Figure 1.1.

Table 1.6: Fields of Contact List Elements

| Field | Data Type | Description |
|---|---|---|
| .uuid | int | unique identifier for each contact list element |
| *.ID* | *int* | *numerical identifier for each contact list element (deprecated)* |
| .x, .y | ints | location of contact in the image in pixels (centroid) |
| .features | array (float) | features used to classify contact |
| .fn | string | filename of image file/some other image ID |
| .side | string | side of vehicle: {'PORT', 'STBD'} |
| .gt | int | ground truth value of contact (if known): 1 = object really is a target; 0 = object really is not a target; empty = ground truth value unknown |
| .type | int | object type (if known): -99 = unknown |
| .class | int | 1 = classified as relevant; 0 = classified as not relevant |

| | | | |
|---|---|---|---|
| **Legend** | Input | Feature extraction | Operator feedback |
| | Detector | Contact correlation | Environmental modules |
| | Classifier | Performance estimation | ATR shell |

(Continued)

| Field | Data Type | Description |
|---|---|---|
| .classconf | float | probability that classification of contact is correct (0 < .classconf < 1) |
| .detector | string | detector identifier |
| .featureset | string | feature set identifier |
| .classifier | string | classifier identifier |
| .contcorr | string | contact correlation identifier |
| .opfeedback | struct | operator feedback (see structure below) |
| .normalizer | string | normalizing algorithm used (empty string if none) |
| .ecdata_fn | string | name of file that contains the remaining contact data |

| | | | |
|---|---|---|---|
| | Input | Feature extraction | Operator feedback |
| **Legend** | Detector | Contact correlation | Environmental modules |
| | Classifier | Performance estimation | ATR shell |

Table 1.7: Fields of .opfeedback Substructure

| Field | Data Type | Description |
|---|---|---|
| .opdisplay | int | operator display mode:<br>0 = don't show this contact<br>1 = show and confirm (likely a target; operator can veto)<br>2 = show and ask (unsure; operator must respond) |
| .opconf | int | operator confidence feedback:<br>1 = likely clutter<br>2 = less likely clutter<br>3 = unsure, could be either<br>4 = less likely a target<br>5 = likely a target |
| .type | int | operator object label (for multi-class classifiers only) : -99 = unknown |

### 1.2.4 Extra Contact Data

As of MATS version 1.1, some of the contact data is written to disk instead of being included in the contact list data structure in order to decrease memory usage. This data is collectively referred to as *extra contact* *data*. To access this data, the binary file designated in a given contact's `ecdata_fn` field must be read. This can be done using the `read_extra_cdata` function, which will return a data structure containing the fields listed in Table 1.8.

Table 1.8: Extra Contact Data Fields

| **Field** | **Data Type** | **Description** |
|---|---|---|
| .uuid | int | unique identifier for each contact list element |
| *.ID* | *int* | *numerical identifier for each contact list element (deprecated)* |
| .hfsnippet | matrix (float) | area around contact in h.f. image (complex) |
| .bbsnippet | matrix (float) | area around contact in b.b. image (complex) |
| .segsnippet | matrix (int) | area around contact in segmentation map |
| .est_vals | matrix (float) | estimator values relevant to .segsnippet: data(:, 1) = indices of regions in .segsnippet data(:, 2:end) = estimator values for regions specified by data(:,1) |
| .lat | float | latitude of contact (in degrees) |
| .long | float | longitude of contact (in degrees) |
| .groupnum | string | ID # of this contact's group; unique for each real object; many images may have objects with the same group # |
| .groupclass | int | 1 = group classified as relevant; 0 = group not classified as relevant |
| .groupclassconf | float | probability that classification of group is correct |
| .groupconf | float | probability that this contact belongs in this group (and represents the same object) |
| .grouplat | float | latitude of group (in degrees) |
| .grouplong | float | longitude of group (in degrees) |
| .groupcovmat | matrix (float) | group lat/long covariance matrix (2x2) |
| .heading | float | nominal heading of the vehicle (in radians) |
| .time | float | timestamp (UTC) |

| | | | |
|---|---|---|---|
| **Legend** | Input | Feature extraction | Operator feedback |
| | Detector | Contact correlation | Environmental modules |
| | Classifier | Performance estimation | ATR shell |

(Continued)

| Field | Data Type | Description |
|---|---|---|
| .alt | float | height of vehicle at contact ping |
| .hf_cres | float | HF image cross-track resolution (in $m$) |
| .hf_ares | float | HF image along-track resolution (in $m$) |
| .hf_cnum | int | HF image cross-track dimension (in pixels) |
| .hf_anum | int | HF image along-track dimension (in pixels) |
| .bb_cres | float | BB image cross-track resolution (in $m$) |
| .bb_ares | float | BB image along-track resolution (in $m$) |
| .bb_cnum | int | BB image cross-track dimension (in pixels) |
| .bb_anum | int | BB image along-track dimension (in pixels) |
| .veh_lats | array (float) | latitude of vehicle |
| .veh_longs | array (float) | longitude of vehicle |
| .veh_heights | array (float) | height of the vehicle at each ping (in $m$) |
| *.spare1-3* | *array (float)* | *spare fields for experimental features* |
| *.spare_str1* | *string* | *spare field for experimental features* |

| | | | |
|---|---|---|---|
| | Input | Feature extraction | Operator feedback |
| **Legend** | Detector | Contact correlation | Environmental modules |
| | Classifier | Performance estimation | ATR shell |

## 1.3 Configuring the GUI

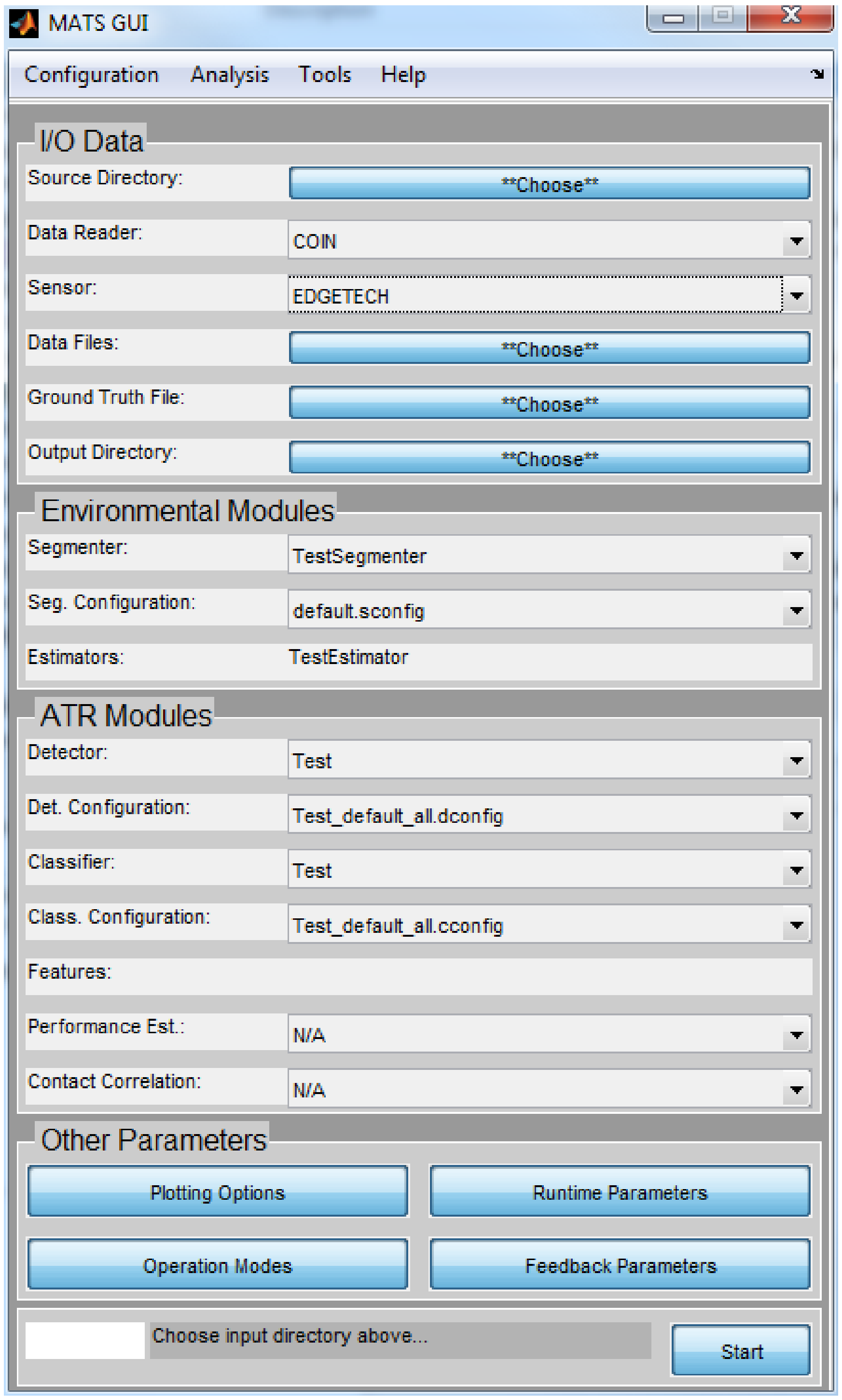
The configurable GUI front end of MATS (`MATS_GUI.m`), shown in Figure 1.4, allows the user to customize the input arguments of the system. It also provides partial control over the run-time process via several adjustable parameters. The items in the main section of the GUI (e.g., ATR Modules) must be configured before beginning the ATR. Each of the buttons in the 'Other Parameters' panel leads to several other options that may be configured if desired; however, their default values should suffice for a simple ATR run. The following sections describe each item in detail.

### 1.3.1 I/O Data

This portion of the GUI specifies which data file(s) to process and where the results should be stored.

#### Source Directory

The source directory is the directory that contains the data files to be processed, either in the directory itself or in its subdirectories. Each data file in the source directory should be of the same file format. This is a



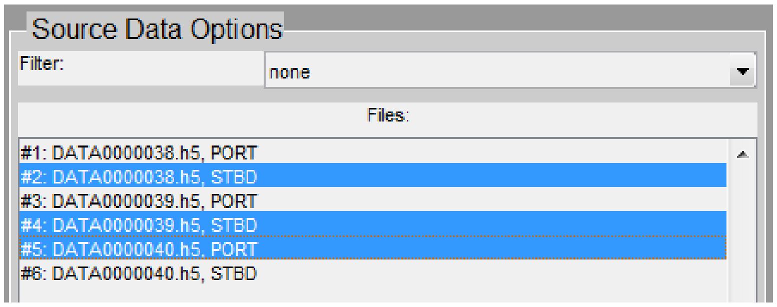
Source Directory: **Choose**
Data Reader: COIN
Sensor: EDGETECH
Data Files: **Choose**
Ground Truth File: **Choose**
Output Directory: **Choose**

Segmenter: TestSegmenter
Seg. Configuration: default.sconfig
Estimators: TestEstimator

Detector: Test
Det. Configuration: Test_default_all.dconfig
Classifier: Test
Class. Configuration: Test_default_all.cconfig
Features:
Performance Est.: N/A
Contact Correlation: N/A

Choose input directory above...

Figure 1.4: MATS configuration GUI


Source Data Options

Filter: none

Files:

#1: DATA0000038.h5, PORT
#2: DATA0000038.h5, STBD
#3: DATA0000039.h5, PORT
#4: DATA0000039.h5, STBD
#5: DATA0000040.h5, PORT
#6: DATA0000040.h5, STBD

Figure 1.5: Source data options GUI

required field, unless using preprocessed detections (as in Section 1.3.5).

### Data Reader Menu

The selected data reader determines how MATS will attempt to read the files within the source directory. Once a data reader is selected, a list of each file within the source directory that matches the formats described in the data readers' description files is created. The length of this list is shown in the 'Data Files' button.

### Sensor Menu

The 'Sensor' selection modifies the run-time parameters of the ATR modules. Module developers should use the `.sensor` field of the input structure to ensure that their algorithms process data correctly. The MATS GUI prevents the user from choosing an algorithm that is incompatible with the selected sensor format according to the `for_sensor` entries of modules' description files. *(Note: See Table 2.4 for more information regarding description files.)*

### Data Files Button

The text of this button displays the number of files in the source directory (or its subdirectories) that are compatible with the current configuration (e.g., data reader, sensor type, etc.). Clicking on this button initializes a new GUI panel that contains additional options for selecting which data files to process. As shown in Figure 1.5, the main feature of this panel is a list of entries representing each compatible data file available. The list can be refined by applying one of the filters in the menu located above it. Any combination of files shown in the list may be selected, and these files will be processed once the ATR processing begins.

### Ground Truth File

The ground truth file contains information relating to the true positions of the objects contained in the source data. Having a ground truth file is not required for basic MATS execution, but it is required for some

MATS analysis functions (e.g., classifier training, ROC curves, etc.).

#### Output Directory

The output directory is where the files created by the ATR processing will be written. This is a required field in all circumstances. *(Note: It is recommended that the output directory be empty before processing begins. When the new output files are saved, they will overwrite any previous results within this directory that have the same filenames. However, there may be additional files left over from the previous run that will interfere with analysis later.)*

### 1.3.2 Environmental Modules

This portion of the GUI configures which environmental modules will be used, if any, during execution. If a valid segmenter module is selected, all available configuration files for that module will appear in configuration menu. Once a segmenter configuration file is selected, the estimator modules that the configuration file uses will be shown in the Estimator item. Currently, all segmenter configuration files must be created manually; the function `sample_seg_config` can be used as a template.

### 1.3.3 ATR Modules

This portion of the GUI configures which ATR modules will be used during execution via several selection menus. Which modules are available depends on the current selections for the data reader and sensor type located above.

The options listed in the configuration file menus are also dependent on the module selections immediately above them. If a module does not have any configuration files for the current conditions, either it must be trained via the detector or classifier training tools, or one of the existing configurations must be modified manually to allow for the desired sensor. Otherwise, the ATR processing cannot begin.

*(Note: The default algorithms (i.e., 'Test') are placeholders that determine results largely at random. They are intended for illustrative purposes only.)*

The feature item indicates which feature modules are included in the selected classifier configuration file. If a new combination of feature modules unrepresented by the existing configuration files is desired, the classifier must be retrained with the desired features.

Performance estimation and contact correlation are optional modules. Selections for both module types default to 'N/A', which disables the module.

### 1.3.4 Plotting Options

Clicking the button labeled 'Plotting Options' creates a new panel containing several options related to displaying sonar imagery:

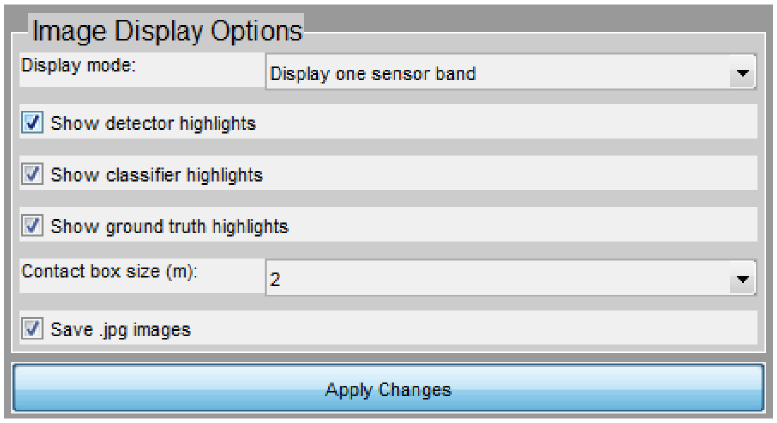


Image Display Options

Display mode: Display one sensor band

[✓] Show detector highlights

[✓] Show classifier highlights

[✓] Show ground truth highlights

Contact box size (m): 2

[✓] Save .jpg images

Apply Changes

Figure 1.6: Image display parameters GUI

### Display mode

This menu determines how many sensor bands will be displayed: one (the primary sensor band, all, or none. (Default: Display one sensor band)

### Show detector highlights

Enabling this option outlines detected contacts with a green box when images are displayed. (Default: enabled)

### Show classifier highlights

Enabling this option outlines contacts that are classified as targets with a red box when images are displayed. These cover the green detection highlights. (Default: enabled)

### Show ground truth highlights

Enabling this option outlines ground truth locations with a blue box when images are displayed. (Default: enabled)

### Contact highlight box size

This menu changes the size of contact highlight boxes that appear around contacts when images are displayed. The selected value corresponds to the height and width of each box in meters. (Default: 2)

### Save .jpg images

Enabling this option saves a copy of each image shown, with desired highlights, into the designated output directory. (Default: enabled)

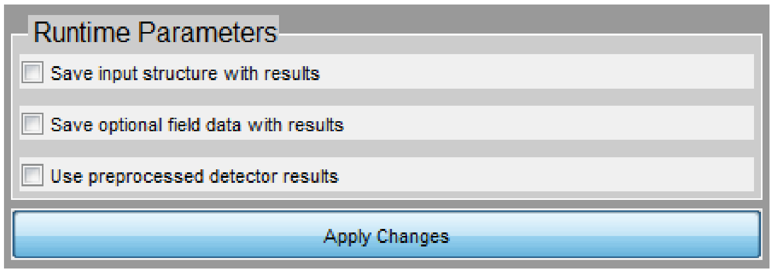

Runtime Parameters
Save input structure with results
Save optional field data with results
Use preprocessed detector results
Apply Changes

Figure 1.7: Runtime parameters option GUI

### 1.3.5 Runtime Parameters

The ‘Runtime Parameters’ button creates a new panel containing several options that affect MATS inputs and outputs:

#### Save input structure with results

Enabling this option saves the corresponding input structure in each output file (`IO_*.mat`). (Default: disabled)

*(Note: This can result in rather large output data files because the source data is essentially being duplicated. As such, it is recommended for use only within a debugging context.)*

#### Save optional field data with results

Enabling this option saves the output of any optional field data in each output file (`IO_*.mat`). (Default: disabled)

#### Use preprocessed detector results

When this option is enabled, the user is prompted for a directory containing previous ATR results at the beginning of the run. The contacts contained in the appropriate output files will be used as detections instead of running the detector to generate them. This will significantly decrease the amount of time required to process an image and can be useful when testing feature sets and classifiers. (Default: disabled)

### 1.3.6 Operation Modes

The options in the ‘Operation Modes’ panel determine which framework variant will be used during execution:

#### Enable distributed mode

Enabling this option allows multiple instances of MATLAB® to run ATR collaboratively. Distributed processing is covered in depth in Section 1.6. (Default: disabled)

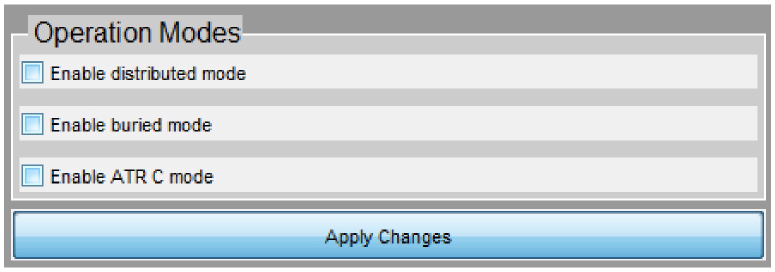
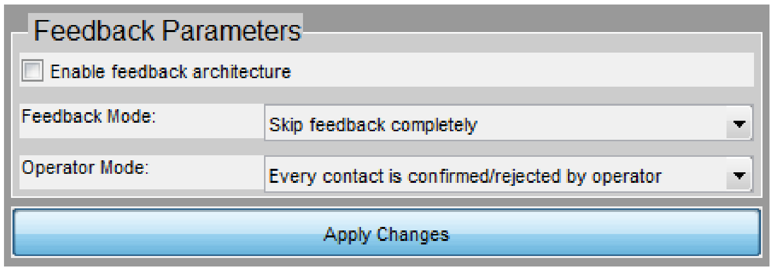


Figure 1.8: Operation modes option GUI



Feedback Mode: Skip feedback completely

Operator Mode: Every contact is confirmed/rejected by operator

Figure 1.9: Feedback parameters option GUI

#### Enable ATR C mode

Enabling this option executes a C framework for ATR (for COIN integration). (Default: disabled)

### 1.3.7 Feedback Parameters

This section has several options that modify how feedback classifiers behave:

#### Enable feedback architecture

This option enable several blocks of code that enable user feedback to modify classifiers. It should usually be enabled when using a feedback classifier, but it is not required; in that case, the classifier will operate in a non-adaptive manner. Enabling the feedback architecture is not recommended for non-feedback classifiers; doing so will only execute unnecessary read/write commands that will slow the processing of data. (Default: enabled for feedback classifiers, disabled otherwise)

#### Feedback mode

This option determines how operator feedback is entered into MATS. (Default: Option #3)

1. 'Use feedback GUI': uses a GUI to view contact snippets, add contacts, and provide operator feedback. (See Section 1.5 (Using the Feedback GUI).)

2. 'Simulate operator feedback w/ archive': uses a saved file containing operator feedback from a prior run. This option should not be used because the archive feature was never fully implemented.

3. 'Skip feedback completely': no feedback input. This is primarily used in situations when some other program (i.e., an external GUI) will collect the feedback from the user, or when it is desired to use a feedback classifier like a static classifier.

4. 'Simulate operator feedback w/ ground truth data': uses ground truth information as a substitute for operator feedback.

#### Operator mode

This option determines how a lack of operator feedback is handled by MATS. (Default: Option #1)

1. 'Every contact is confirmed/rejected by operator': the operator cannot skip contacts

2. 'Interpret no operator comment as implicit agreement': when an operator skips a contact, he agrees with the ATR classification

3. 'Use only explicit operator calls': when an operator skips a contact, no feedback will be used.

### 1.3.8 Run-time Elements

The bottom-most region of the MATS GUI is composed of a start button, a progress bar, and a message area.

The start button is initially disabled and will remain unclickable until inputs and outputs are specified and all modules have valid configurations. The remaining items in the GUI have usable default values.

The main function of the message area is to track the progress of the run along with the progress bar. However, before the run begins, it displays messages to help guide the user toward a complete configuration.

### 1.3.9 Menu Options

The GUI has several other features that can be accessed through the menu at the top of the MATS GUI. Some tools like module training can help during MATS configuration; Others are analysis tools that can only be used after ATR execution.

#### Save/Load Configuration

The GUI allows a run-time configuration to be saved for later use via **Configuration** → **Save** in the GUI menu. Loading a configuration can be done similarly via **Configuration** → **Load**. This feature can be used to establish a consistent run-time environment.

```
        ImageHF-10-7, PORT


  C#163 @ ( 400, 301) -------- ......No match......
x C#164 @ (2401, 650) -------- GT#122 @ (2401, 650)
x C#165 @ (3208,1151) -------- GT#123 @ (3201,1150)
x C#166 @ (2022,1401) -------- GT#124 @ (2001,1400)
........No match..... -------- GT#121 @ (1601, 900)
```

Figure 1.10: Excerpt from a sample ground truth analysis

### Refresh Module Data

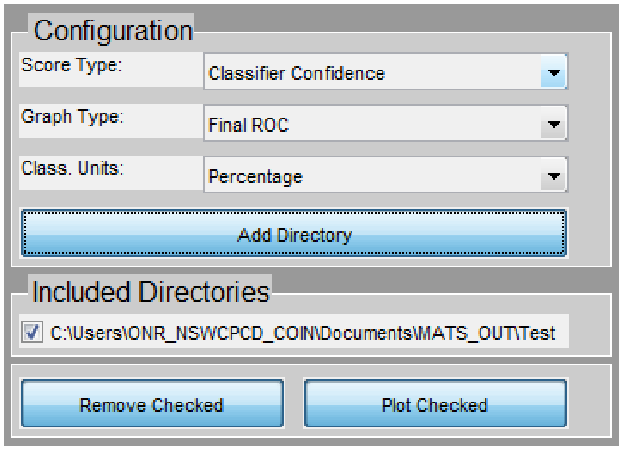
The GUI has a master list of all available modules and module configuration files. This list occasionally needs to be updated via **Configuration** → **Refresh Module Data**. For example, a newly added module will only appear in the appropriate selection menu after this step has been performed.

### Launch MATS Slave

Selecting **Configuration** → **Launch MATS Slave** will execute the MATS slave function on this instance of MATLAB®. Note that in order for this to accomplish anything, a MATS master process must also be running elsewhere on the network. *(Note: See Section 1.6 (Using Distributed Processing Mode) for more information about MATS' distributed mode.)*

### Ground Truth Analysis

Selecting **Analysis** → **Compare Contacts To Ground Truth** launches the ground truth analysis tool. Given both a directory containing `IO_*.mat` output files and a ground truth file, all contacts in that directory are loaded into memory and their locations are compared to the entries listed in the selected ground truth file. The final printout organizes the data such that each group of text corresponds to one input image; a sample excerpt of this process is shown in Figure 1.10. When a ground truth object matches a contact, the location of both entries will appear in the same line of text. Any unmatched ground truth entries will appear at the end of the group corresponding to the image in which it is located. Contacts that were classified as targets are indicated by an `X` at the beginning of the line. Finally, a complete list of all unmatched ground truth entries located in imagery that was actually processed is included at the end of the analysis along with some summary statistics.



Score Type: Classifier Confidence
Graph Type: Final ROC
Class. Units: Percentage
☑ C:\Users\ONR_NSWCPCD_COIN\Documents\MATS_OUT\Test

Figure 1.11: ROC curve configuration menu

### ROC Curves

Receiver operator characteristic (ROC) curves for one or more ATR runs can be generated by selecting **Analysis → Generate ROC Curve** from the menu. The resulting GUI, shown in Figure 1.11, allows configuration of the type of curve produced and the axes' units. To add a ROC curve to the figure, click the 'Add Directory' button and select the output directory of an ATR run. The directory's path then appears below, in the panel titled 'Included Directories'. Curves may be deleted via the 'Removed Checked' button, at which point all checked directories will be removed from the panel.

This tool can create several types of curves. The 'Final ROC' option plots one curve using all of the relevant contacts generated during the ATR run. The 'Incremental ROC' option creates a sequence of plots, with each one incorporating an additional image's worth of data; these plots are used as frames to create an animation that shows how the classifier's performance changes over time as more data is encountered.

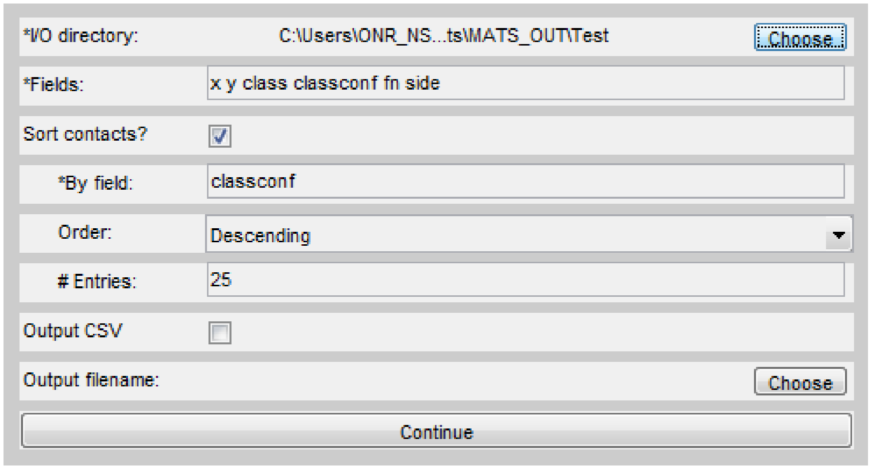
*(Note: All ROC curve-related functionality requires the MATLAB® Statistics Toolbox, the 'perfcurve' function in particular.)*

### Confusion Matrix

A confusion matrix can be generated for a selected run via **Analysis → Generate Confusion Matrix**. This will summarize the numbers of correctly identified objects, correctly rejected non-objects, misses, and false alarms. *(Note: This requires the MATLAB® Statistics Toolbox.)*

### Ground Truth Review Tool

The Ground Truth Review Tool (**Analysis → Review Ground Truth**) provides a mechanism for examining and modifying entries in a ground truth file. Each row in the table represents an entry in the file and corresponds to a particular location within a particular image. Clicking on an entry displays the image



| | |
|---|---|
| *I/O directory: | C:\Users\ONR_NS...ts\MATS_OUT\Test |
| *Fields: | x y class classconf fn side |
| Sort contacts? | ✓ |
| *By field: | classconf |
| Order: | Descending |
| # Entries: | 25 |
| Output CSV | ☐ |
| Output filename: | |

Figure 1.12: Section of the Contact Data Display Tool

in which it is located and marks its position in the image. Entries may be toggled on and off using the checkboxes in the leftmost column. If the file is saved, then all unchecked entries will be disabled and ignored by MATS.

### Contact Data Display Tool

The Contact Data Display Tool (**Analysis** → **Display Contact Data**) provides a convenient way to display contact data in a formatted table. The tool's configuration GUI (Figure 1.12) allows the user to customize which data fields will be included in the table and how the contact data will be organized. Two preset configurations can be accessed quickly via its menu; each one will display all contacts' locations (in either pixel or geospatial coordinates), filenames, side information, classifications, and classifier confidences. Alternatively, the parameters can be customized to include any combination of the contact data fields that are not large vectors; these are excluded due to the infeasibility of displaying large vectors in a tabular format.

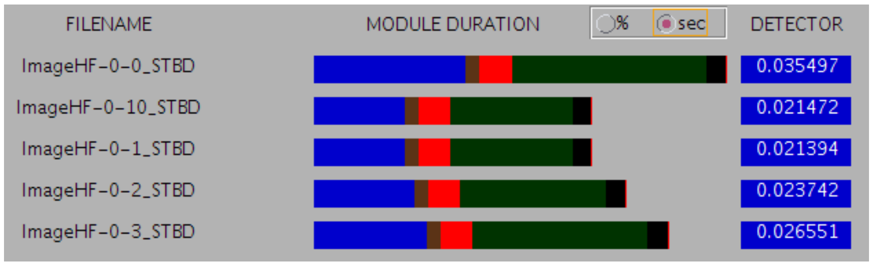
Once a directory containing contacts and the desired fields are selected, the formatted table can be generated by clicking the button at the bottom of the GUI. The smaller contact fields from all of the selected files will be aggregated into a master contact list; depending on the number of contacts in the folder, this step may take some time. The resulting table (Figure 1.13) will be shown on the main MATLAB® prompt, but a copy may also be saved to an output file if desired by choosing an output file name in the tool's configuration GUI before execution.

### Timing Analysis Tool

The Timing Analysis Tool (**Analysis** → **Timing Analysis**) displays the elapsed time of major code segments for each of the files in the selected output directory. A portion of this display is shown in Figure 1.14.

```
============================================================
|                  UUID                  |  X   |  Y  | CLASS |
============================================================
| A64478FE-51FE-4CC0-A224-788CAEE54880 | 3788 | 172 |   0   |
| 5D10E1E0-B748-4531-A4A5-9F6AA9B63AC5 | 2405 | 190 |   1   |
| 1BF57CBF-99F9-41BD-8F0B-17701684EF15 | 4518 | 221 |   0   |
| 31D51D24-1D24-4543-BE71-B2689154297C |  575 | 271 |   0   |
| 3C5D844E-F997-4E5D-BEAB-582AEE0BA623 | 2531 | 765 |   0   |
============================================================
```

Figure 1.13: Sample contact data



| FILENAME | MODULE DURATION (% [✓ sec]) | DETECTOR |
|---|---|---|
| ImageHF-0-0_STBD | | 0.035497 |
| ImageHF-0-10_STBD | | 0.021472 |
| ImageHF-0-1_STBD | | 0.021394 |
| ImageHF-0-2_STBD | | 0.023742 |
| ImageHF-0-3_STBD | | 0.026551 |

Figure 1.14: Section of the Timing Analysis Tool

Each file name is listed along with the individual module totals. The colored bars in the center of the display show the modules' execution time, measured either in seconds or as a percentage of the total time, depending on the radio button selected.

### Data Comparison Tool

The Data Comparison Tool (**Analysis** → **Compile Comparison Data**) compiles the results for one or more runs, measures the differences in classification matches and confidences between the initial baseline run and subsequent runs, and stores the results in a comma-separated value (.csv) file that can be opened as a spreadsheet in a program like Microsoft Excel. The totals listed at the bottom of the spreadsheet give an indication of how much the classifier results have changed relative to the baseline run. While this tool is intended to analyze feedback classifiers, it may also be used to compare different classifiers so long as all of the contact locations are the same (i.e., the 'Use Preprocessed Results' option has been used).

### Performance Report

The report generated by **Analysis** → **Generate Performance Report** gives a concise overview of a particular set of ATR results. The report includes critical input parameters as well as many of the performance metrics described in earlier sections.

*(Note: This feature requires LaTeX to be properly installed and configured to produce the report.)*

### Classifier Training

MATS also has mechanisms in place to facilitate classifier training via **Tools** → **Train Classifier**. Given the ATR results for a data set, the GUI can extract the features and ground truth labels from the set's contacts and pass them into a training function. *(Note: See Section 2.2.5 (Training Functions) for more information about what this entails.)*

A GUI has been provided to assist in the training process, as shown in Figure 1.15. A set of MATS results is required for training data. If no other classifier configuration files are available, the Test classifier can always be ran to generate these results. In addition, a classifier must be selected from the 'Classifier' selection menu and at least one feature module must be selected in order to begin classifier training.

Features are selected by checking the box to the left of the feature module name. Clicking on the feature module name from the list will bring up the feature module's options GUI, if options are available. Closing the options GUI will return to the Classifier Training GUI.

The training process is started by clicking the 'Start' button at the bottom of the GUI. **Training can take a considerable amount of time and memory** depending on the modules selected and the size of the training set. Once the training process has finished, a classifier configuration file (`*.cconfig`) is generated for the selected classifier using the chosen parameters. This configuration file is stored in the classifier's module directory. The training file is detected by MATS after restarting the GUI or by selecting 'Refresh Module Data' from the 'Configuration' menu of the MATS GUI.

### Documentation

This document may be accessed via MATS by selecting **Help** → **MATS Documentation** from the GUI menu.

## 1.4 Running MATS

Once all of the run time requirements have been met, clicking the start button will initiate a loop that processes each of the selected images in turn. For each image, status messages printed to the MATLAB® prompt provide progress updates. These messages can help the user monitor the progress of an individual data file through the ATR. In addition, the progress bar located at the bottom of the configuration GUI advances as each files is processed.

When an image has completed processing, the results are saved in a file called `IO_[INPUT_FNAME].mat` where `[INPUT_FNAME]` is the name of the source data file without its extension. Similarly, smaller files

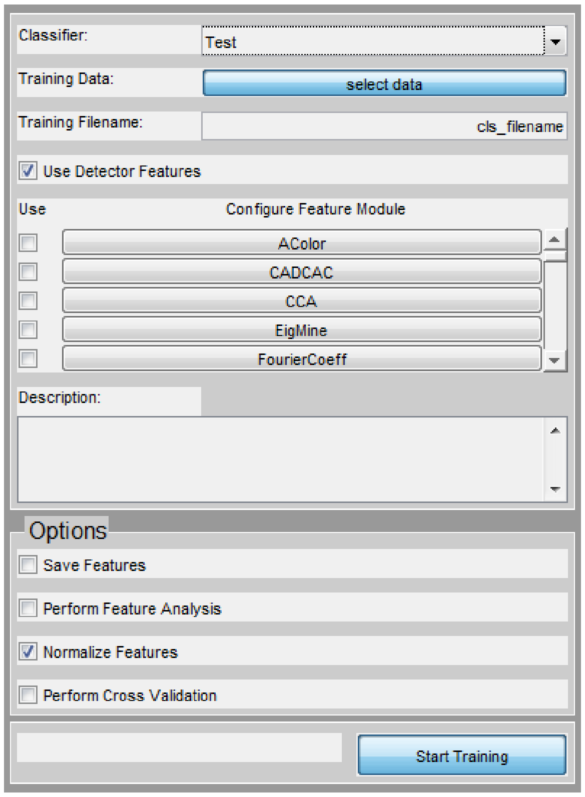


Figure 1.15: The classifier training GUI

`ROC_[INPUT_FNAME].mat` and `TIME_[INPUT_FNAME].mat` will be created that can be used in the ROC curve and timing GUI tools, respectively. If enabled, the image will be displayed with colored boxes highlighting contact and/or ground truth locations; a copy of this image will also be saved in the output directory.

If an image is too large, it will be split into smaller chunks for the detector portion of the processing. This step is invisible to the other modules, and there is nothing extra that must be done to make a detector compatible with the chunking process. However, it should be noted that if chunking is necessary, then the output images will be chunked as well.

After all of the selected images have been processed, contact correlation is run if it was enabled in the GUI configuration. Finally, if a ground truth file had been entered into the GUI before execution, the ground truth analysis function prints a summary of the final contact list and ground truth information. The ground truth analysis function is also discussed in more detail in Section 1.3.9.

## 1.5 Using the Feedback GUI

The feedback GUI provides an interactive way for the user to provide classifier feedback. This GUI is used whenever the `Use feedback GUI (series)` option is selected in the Configuration GUI. *(See Section 1.3.7 for more information.)*

The feedback GUI has three main components. The right section displays the full image with several types of highlight boxes. Detections, classifications, and ground truth entries are indicated by green, red, and blue boxes, respectively, like the regular image display. In addition, manually added contacts are magenta and the current contact is white.

To the left is a smaller image of the HF snippet taken around the current contact. The current contact is either the next contact in the list that is awaiting operator feedback, or the last place that the operator has clicked in the image.

Below the snippet are several buttons that generate and record feedback information. When using binary classifiers, once one of the feedback rating radio buttons has been selected, the selected rating will be recorded after either clicking the `Next` button (if rating an existing contact) or the `Add` button (if manually adding a new contact). In the multiclass case, an object type menu is also available; the default value will be for the class that the classifier has chosen, but this can be changed by the user before recording the feedback. After viewing a manually selected snippet, the current contact can be reset to be the next contact in the list that is awaiting feedback by clicking the `Revert` button.

## 1.6 Using Distributed Processing Mode

MATS includes the capability to allow groups of MATLAB® instances to collaboratively process a common data set. These instances may be running on the same computer or on separate, networked computers; as long as each instance can access the common data files used, it does not matter which is the case.

### 1.6.1 Overview of Architecture

The distributed processing architecture introduces several new components and files. These are summarized here and will be described in more detail in the following sections:

- The ***MATS GUI*** configures individual ATR tasks in the same manner as before. However, when operating in distributed mode, the configuration is added to the bulletin file and is eventally passed into the ATR master process instead of being passed directly to the main ATR loop.
- An ***ATR Task*** is a batch process that runs ATR on a collection of image data. This is identical to an ordinary, batch mode ATR run, except that in this context it is possible for multiple processes to collaborate on a task in order to finish it more quickly.
- The ***Common Root*** is the outermost directory to which all participating components are assumed to have access. This directory contains the bulletin file and input and output subdirectories for each ATR task.
- The ***Bulletin File*** contains a complete list of ATR tasks that form the task queue. Each entry includes the basic information (e.g., input and output subdirectories) that defines a task.
- The ***Task Manager GUI*** displays a list of ATR tasks that are currently listed in the bulletin. Users can get information about a given task, abort a task, and start processing the ATR tasks currently listed in the bulletin.
- The ***ATR Master Process*** is the primary ATR processing function in distributed mode. It is responsible for managing both ATR tasks and a collection of ATR slave processes working on those tasks.
- ***ATR Slave Processes*** are instances of a subordinate ATR function that process whatever ATR jobs the master process assigns to them until all of the tasks listed in the bulletin have been completed.
- A ***Task Progress Monitor GUI*** provides a visual summary of the state of an ATR task, including its total progress and which slave processes are currently working on which data files.

#### MATS GUI

When distributed mode is enabled via the checkbox in the operation modes panel, the start button at the bottom of the GUI becomes an 'Add to queue' button; clicking this will add a copy of the current GUI configuration to the bulletin and initialize (or update) the Task Manager to include the newly generated task. This will occur even if the ATR is currently running.

### ATR Task Manager

The ATR Task Manager lists all of the tasks that are currently in the bulletin. In particular, the final subdirectory of the output path (as originally specified in the MATS GUI) and the selected ATR modules are used to give an overview of each task. Clicking the 'View Params' button will bring up a list of all of the MATS runtime parameters and their values. The button at the bottom of the window spawns an instance of the ATR master processes, thus starting the ATR processing. At this point, tasks may still be aborted by clicking the 'Abort' buttons on the right side of the task manager window.

### Common Directories and Files

The common directories and the files that they contain are the primary means by which the ATR processes communicate with each other. The common root directory is the outermost directory that is assumed to be shared among the master and slave processes. This directory contains the bulletin file as well as all of the communications between the master and slave processes.

The common root directory also contains an input directory and an output directory for each task. The input directory contains the code of all ATR modules to be used in addition to task-specific data. The output directory is used to save the ATR results that are produced by the ATR slave processes, functioning in the same manner as the output directory for a standard ATR batch process.

*(Note: Currently the output directory that is specified in the MATS GUI is largely ignored in distributed mode. Only the last directory segment of the selected output directory is used to name the task folders in the common root directory, where the tasks' results are stored.)*

### Source Data

Source data files can be located in one of two places. They are normally found in the directory that was specified using the 'Source directory' field in the main MATS GUI. However, if a source data directory is not specified in this manner, then the common input directory is used instead.

Currently, all source data files must be present in the source data directory (specified via the MATS GUI) at the beginning of execution. At this point, the master list of files to be processed is generated. Any files that are added after the processing has started will be ignored.

### ATR Master Process

The ATR master process is responsible for managing the completion of all ATR tasks that are specified in the bulletin. After initializing all of the open tasks, the master process cycles through them, staying up to date on the progress of each task by searching for new data files and completed jobs, and assigning any idle workers a new job to complete.

#### ATR Slave Processes

The ATR slave process is very straightforward. Once initialized, it waits for its job assignments to be written in the common root directory. Then, it processes whatever data it has been assigned using the modules saved in the current task's input directory and saves the results in that task's specified output directory. This process repeats itself until the master process indicates that it no longer needs this slave's assistance. An individual slave may work on several different ATR tasks before finally being released.

#### Type of Jobs

The master process can issue several types of jobs, which allow the slave to perform different roles. There are currently two types of jobs that have been implemented. The most common type is an *ATR job*, which runs the specified ATR modules on the specified data. This is analogous to a call to the standard batch mode ATR processing function (`mats_proc_loop`).

The other type is a *feedback job*, which monitors a task's output directory and applies feedback (currently derived only through ground truth) to the contacts. A slave that is assigned a feedback job will not be assigned any other jobs until the associated task has completed and all of the task's feedback has been applied. The current implementation supports only one slave serving the feedback role. The master will automatically assign a feedback job only to the second slave that it assigns to a feedback task.

#### Task Progress Monitor GUI

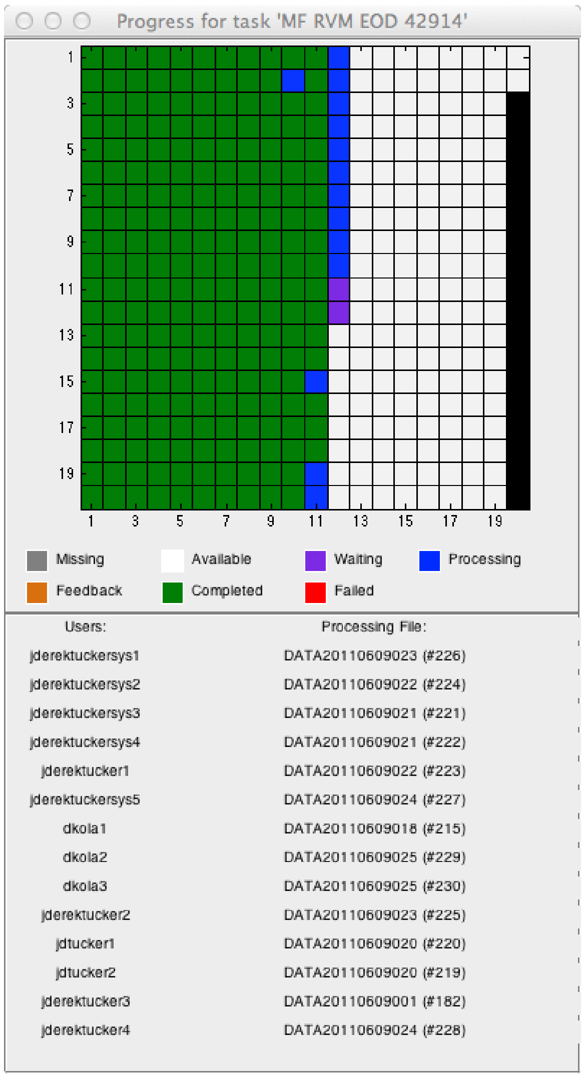
A task's progress monitor (shown in Figure 1.16) summarizes the state of the task at a given time. There are two main sections to this display. The upper portion is a colored grid, with each cell's color indicating the status of a particular file to be processed. The lower portion lists all of the available slaves currently working on the task and which image they are currently processing. Both sections will be updated periodically.

The transitions between the upper portion's status colors are illustrated in Figure 1.17. A grid cell remains *gray* until the master discovers the presence of the corresponding source data file, at which point it becomes *white*. Once a slave is assigned to process this data, the cell turns either *blue* (if the job is assigned to the slave directly) or *purple* (if it is instead created as part of the job reserve). If feedback is enabled, then the cell will shift to an intermediate *orange* state, which indicates that the master is waiting for feedback. Finally, the cell will turn *green* if the ATR was completed successfully, or *red* if an error occurred.

### 1.6.2 Distributed ATR Processing via MATS

#### Initializing the Master

For each task, $T_i$, the main MATS GUI is used to configure the various parameters of the task, namely which data to process and which modules to execute during the ATR processing. Once the configuration is



| | |
|---|---|
| 🟫 | Missing |
| ⬜ | Available |
| 🟪 | Waiting |
| 🟦 | Processing |
| 🟧 | Feedback |
| 🟩 | Completed |
| 🟥 | Failed |

| Users: | Processing File: |
|---|---|
| jderektuckersys1 | DATA20110609023 (#226) |
| jderektuckersys2 | DATA20110609022 (#224) |
| jderektuckersys3 | DATA20110609021 (#221) |
| jderektuckersys4 | DATA20110609021 (#222) |
| jderektucker1 | DATA20110609022 (#223) |
| jderektuckersys5 | DATA20110609024 (#227) |
| dkola1 | DATA20110609018 (#215) |
| dkola2 | DATA20110609025 (#229) |
| dkola3 | DATA20110609025 (#230) |
| jderektucker2 | DATA20110609023 (#225) |
| jdtucker1 | DATA20110609020 (#220) |
| jdtucker2 | DATA20110609020 (#219) |
| jderektucker3 | DATA20110609001 (#182) |
| jderektucker4 | DATA20110609024 (#228) |

Figure 1.16: Distributed processing progress GUI.

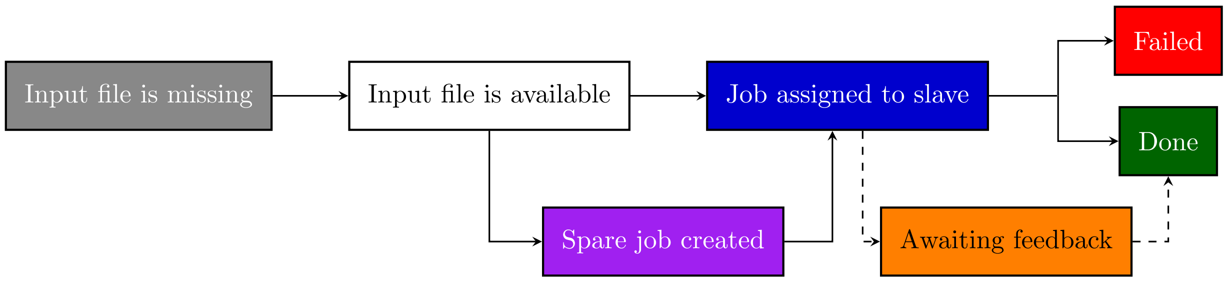


Figure 1.17: Task Progress File States

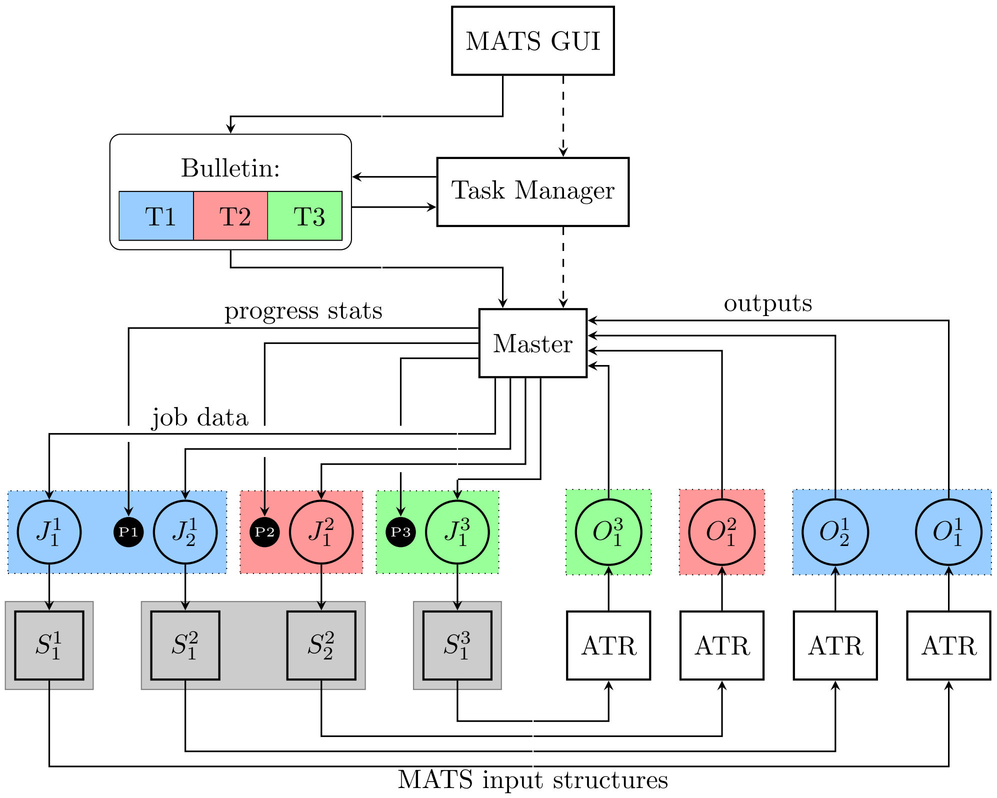


Figure 1.18: Architecture of MATS in distributed mode.

complete, the task is added to the bulletin via the 'Add to Queue' button at the bottom of the GUI when distributed mode is enabled throught the menu options. *(Note: See Section 1.3.9.)*

This launches the Task Manager GUI, which will show the newly added task. Additional tasks will appear as they are added via the MATS GUI. A task can also be removed from the task manager by aborting the task. The 'Begin Processing' button at the bottom of the Task Manager GUI will launch the master process that manages the tasks listed in the bulletin. These components are shown in the top half of Figure 1.18. At this point there are not any workers to which jobs can be assigned, so the master process becomes idle shortly.

### Initializing the Slaves

Each machine, $M_k$, that will contribute to the ATR processing must start at least one MATLAB® instance and run the function `mats_slave`, which has one optional input argument: the path of the common root directory on that machine. If this argument is not specified, the user will be asked to provide it. The $n$-th instance on machine $M_k$ is denoted by $S_n^k$ in Figure 1.18. This example has four slaves running on three machines: one dedicated solely to the first task ($S_1^1$), one dedicated solely to the third task ($S_1^3$), and one whose resources are split across the first and second tasks ($S_1^2$ and $S_2^2$). Once an ATR slave has been initialized, it waits for the master process to assign it a job.

### The Processing Cycle

Once the master process recognizes that an ATR slave is available, it will assign the slave (the $q$-th process working on task $T_i$) the job $J_q^i$ by saving all of the information that is required to run the ATR to a specified filename within the common directory. The slave watches for this job file, and once the file is detected, it passes the MATS input structure contained in the job file into the ATR. The ATR function used is similar to the standard MATS version, except that it uses copies of the chosen modules that are stored in the task's common directory to ensure that all slave processes use identical versions of the ATR modules.

When the ATR is finished, the ATR outputs are saved in a corresponding output file, $O_q^i$, in the common output directory. The master process monitors the common output directory for new files. When a new file is discovered, the master process determines which process was working on that job and assigns that newly idle slave another job, if possible. This process repeats until the master indicates that all of the jobs for this task have already been processed.

At this point, the slaves that were previously assigned to the newly finished task will look in the bulletin for any outstanding ATR tasks. If there are still unfinished tasks, each slave will reinitialize itself and begin work on the next unfinished task.

## 1.7 Frequently Asked Questions

1. *I see messages saying that 'bkuplock.txt' and/or 'bkupedit.txt' cannot be opened. Is that bad?*

   Probably not. At the beginning of the run, `bkuplock.txt` and `bkupedit.txt` will not exist. `bkuplock.txt` will not be written to until an operator feedback action takes place. If you haven't submitted feedback on any contacts yet, messages indicating that those files do not exist or cannot be opened are expected.

2. *Why do the detections as measured by the Ground Truth Analysis Tool differ from the Confusion Matrix Tool?*

   These two tools are actually measuring slightly different things. The Ground Truth Analysis Tool measures with respect to ground truth entries. The detection rate calculated by this tool is the percentage of the entries listed in the ground truth file that were found by the detector; the numerator of this fraction is the *total number of ground truth entries that correspond at least one contact* (i.e., a detection). In contrast, the Confusion Matrix Tool focuses on contacts. The left column in the matrix represents all contacts that should be classified as an object. The sum of this column is the *total number of contacts that correspond to at least one ground truth entry.* This is the reverse of the former case. While these two totals are often quite similar, they may not be identical if, for example, multiple contacts are generated for the same ground truth entry.

# Chapter 2

# Developer Reference

## 2.1 Adding a Custom Module Class to MATS

MATS v3.2 introduces a new module paradigm based on a class hierarchy. The environmental modules introduced in v3.2 (i.e., image segmenters and parameter estimators) use this new class hierarchy, whereas the non-environmental modules (i.e., all other modules) continue to use the original paradigm. While there are similarities between these two groups, they will be presented in isolation for clarity. This section will focus on the newer paradigm as implemented by the environmental modules.

### 2.1.1 Common Requirements

#### Copying Code into the Directory Structure

A new module must reside in the proper location within the MATS directory structure in order to be recognized by MATS. Each module pool (e.g., segmenters) has a dedicated subdirectory that contains all of the modules of that pool. Within that directory, each module's code is contained in a subdirectory of its own called the *module root directory*; this must be created for the new module. The general form for a module root directory path is `[MATS_ROOT]\[POOL_SUBDIR]\[MODULE_SUBDIR]`, where `[MATS_ROOT]` is the root directory of the MATS package (i.e., the directory that contains the GUI), and `[MODULE_SUBDIR]` and `[POOL_SUBDIR]` are subdirectory strings for an individual module and its corresponding pool, respectively. Pool subdirectories for each module type are listed in Table 2.1. `[MODULE_SUBDIR]` will be used to identify the algorithm throughout the GUI, so it should be fairly short, but still descriptive. For example, the module root directory of the TestSegmenter image segmenter is `[MATS_ROOT]\Environ\Segmenters\TestSegmenter`.

#### Module Class Definition

The module's class definition must be of the form `[MODULE_SUBDIR].m`, where `[MODULE_SUBDIR]` is the module root subdirectory string as before. For example, the included TestSegmenter segmenter's main function is `TestSegmenter.m`. The class definition must be located in the module's root directory; otherwise, the module will not be identified during initialization and as a result will not appear in the GUI's detector selection menu.

All module classes must extend their module pool class, which in turn extend the `MATSModule` superclass. `MATSModule` contains very little, but it does ensure that all module objects are configured before they are used. The method `configure(...)` should be called to configure algorithm modules, but the actual implementation should be in the method `configure_(...)`. The reason for this is that `configure(...)` will ensure that the flag indicating successful configuration is properly set.

Table 2.1: Summary of Integration Parameters for Module Classes

| **Type of Module** | **Pool Subdirectory** |
|---|---|
| Image Segmenter | `Environ\Segmenters\` |
| Parameter Estimators | `Environ\Estimators\` |

### 2.1.2 Additional Requirements for Image Segmenters

#### ImageSegmenter Pool Class

All image segmenters should extend the `ImageSegmenter` pool class. This class contains one additional property: (`.estimators`), a list of the estimator modules that this segmenter will use to segment the imagery. These estimators will be automatically created and configured during the segmenter configuration step. Additional properties may be added to individual image segmenter modules (e.g., `.num_bins` in `TestSegmenter`) as needed.

`ImageSegmenter` also contains the abstract method `segmentImage(...)`, which takes in a MATS input structure and returns two outputs: a segmentation map and a matrix containing the value of each configured estimator for each segmentation region. A module's implementation of this function will be invoked during run-time.

#### Segmenter Configuration Files

At least one *segmenter configuration file* must be created before a segmenter can be run successfully. Each of these files contains segmenter parameters that may change over different sensors and environments. The data is stored in the fields of a structure, as summarized in Table 2.2. All configuration files must be saved in a `.mat` format with a `.sconfig` file extension. The configuration that is chosen from the GUI's 'Seg.

Table 2.2: Fields of a Segmenter Configuration File

| **Field Name** | **Description** |
|---|---|
| `.uuid` | unique identifier string |
| `.note` | string describing configuration |
| `.time` | time stamp (from MATLAB®'s `now` function) |
| `.segmenter` | string containing segmenter name |
| `.sensor` | sensor string (should match one of the strings in the GUI's dropdown menu) |
| `.segver` | version of segmenter |
| `.seglen` | number of values returned by segmenter |
| `.segcfg` | structure for other segmenter parameters |
| `.estimators` | cell array of strings containing estimator names |
| `.estvers` | version of each estimator |
| `.estlens` | number of values returned by each estimator |
| `.estcfgs` | array of estimator configuration structures (passed to estimators) |

Configuration' selection menu will be used at run time. *(Refer to Section 1.3.3 for more information about the GUI).*

### 2.1.3 Additional Requirements for Parameter Estimators

#### Estimator Pool Class

All parameter estimators should extend the `Estimator` pool class. This class contains one additional property: (`.num_values`), the length of the vector of values that the estimator will return. `Estimator` also contains the abstract method `estimateRegion(...)`, which determines the parameters for a single region defined by a binary mask. A module's implementation of this function will be invoked during run-time either directly or through the method `estimateAllRegions(...)`, which loops over for each region in a segmentation map.

## 2.2 Adding a Custom Module to MATS

This section describes how to integrate the older modules into MATS' original module framework. In MATS v3.2, this means all modules except image segmenters and parameter estimators. Throughout this section, references to "all modules" should be interpreted as the older set of modules that existed prior to MATS v3.2 (i.e., all modules except image segmenters and parameter estimators, which were already covered in Section 2.1). The integration process is similar for all [pre-v3.2] module types, although there are some notable differences that will be covered in more detail in the subsequent sections.

### 2.2.1 Common Requirements

#### Copying Code into the Directory Structure

A new module must reside in the proper location within the MATS directory structure in order to be recognized by MATS. Each module pool (e.g., detectors) has a dedicated subdirectory that contains all of the modules of that pool. Within that directory, each module's code is contained in a subdirectory of its own called the *module root directory*; this must be created for the new module. The general form for a module root directory path is `[MATS_ROOT]\[POOL_SUBDIR]\[MODULE_SUBDIR]`, where `[MATS_ROOT]` is the root directory of the MATS package (i.e., the directory that contains the GUI), and `[MODULE_SUBDIR]` and `[POOL_SUBDIR]` are subdirectory strings for an individual module and its corresponding pool, respectively. `[MODULE_SUBDIR]` will be used to identify the algorithm throughout the GUI, so it should be fairly short, but still descriptive. For example, the module root directory of the Test detector is `[MATS_ROOT]\Detectors\Test`.

The module's main function must be of the form `[PREFIX]_[MODULE_SUBDIR].m`, where `[MODULE_SUBDIR]` is the module root subdirectory string as before and `[PREFIX]` is a prefix that corresponds to the module pool type as specified in Table 2.3. For example, the included Test detector's main function is `det_Test.m`. The main function must be located in the module's root directory. Otherwise, the module will not be identified during initialization and as a result will not appear in the GUI's detector selection menu.

Table 2.3: Summary of Integration Parameters

| **Type of Module** | **Pool Subdirectory** | **Prefix** | **Need Desc. File?** |
|---|---|---|---|
| Data Reader | `Data␣Readers\` | `read` | X |
| Detector | `Detectors\` | `det` | X |
| Feature Generator | `Features\` | `feat` | X |
| Classifier | `Classifiers\` | `cls` | X |
| Performance Estimation | `Performance␣Estimation\` | `perf` | |
| Contact Correlation | `Contact␣Correlation\` | `cor` | |

#### Creating a Description File

Furthermore, each module has a *description file* called `about.txt` located in its root directory. This file describes what is required for the successful execution of the module in the manner shown in Figure 2.1. For performance estimation and contact correlation modules, the description file is optional but should be included for consistency. When creating the module data structure during initialization, the GUI parses this file and extracts the necessary information from it. When this module is selected via the configuration GUI, the run-time configuration will be modified appropriately. All of these options are summarized in

Figure 2.1: Sample Description File

```
module_name: Test Classifier

module_tag: TEST

uses_feedback: 0

multiclass: 0
```

Table 2.4. Fields that are specific to one particular module type may have more information about them in the following subsection for that module type. For example, regular expression fields used by data readers must be compatible in ways described in Section 2.2.2.

Table 2.4: Description File Entries Required for Module Pools

| **R** | **D** | **F** | **C** | **S** | **E** | **Description File Entry** |
|---|---|---|---|---|---|---|
| X | X | X | X | X | X | `module_name`: character string indicating the name of the module. |
| X | X | X | X | X | X | `module_tag`: short character string (3-4 alphanumeric characters). |
| X | X | X | X | X | X | `version`: version string (e.g., '`X.Y`'). |
| X | | | | | | `field_names`[1]: list of N field names of frequency bands present in data (e.g., `hf; bb`) |
| X | | | | | | `band_regexps`[1]: list of N regular expressions used to separate list of files into frequency band groups (as specified by `field_names`) |
| X | | | | | | `band_replace_strs`[1]: list of N strings to replace `band_regexps` when searching for a file containing data of a different frequency over the same geographic area |
| X | | | | | | `port_regexp`[2]: regular expression used to identify port side files |
| X | | | | | | `stbd_regexp`[2]: regular expression used to identify starboard side files |
| | X | | | X | X | `for_sensors`: strings indicating which sensors that the module can process. (Matches items in sensor selection menu in GUI) |
| | X | | | | | `for_buried`: binary flag[3] indicating that the module can detect buried mines. (Used to filter detector list in main GUI) |
| | | X | | | | `reqs_inv_img`: binary flag[3] indicating whether this feature extraction method requires inverse imaging. |
| | | X | | | | `reqs_bg_snippet`: binary flag[3] indicating whether this feature extraction method requires background snippets. |

| **Legend** | Data Reader | Detector | Feature | Classifier | Segmenter | Estimator |
|---|---|---|---|---|---|---|

(Continued)

| R | D | F | C | S | E | Description File Entry |
|---|---|---|---|---|---|---|
| | | X | | | | `feat_mode`: string indicating how features are handled. `append` adds onto the existing features, if any were created by the detector; `overwrite` overwrites them. |
| | | | X | | | `uses_feedback`: binary flag[3] indicating whether this classifier can adapt using operator feedback. |
| | | | X | | | `multiclass`: binary flag[3] indicating whether this classifier can distinguish among object types (1) or only between object and non-object (0). |

| **Legend** | Data Reader | Detector | Feature | Classifier | Segmenter | Estimator |
|---|---|---|---|---|---|---|

### Refreshing Module Data

During initialization of the MATS GUI, all available ATR modules located within the MATS directory are automatically imported into the module data structure. Modules and configuration files that have been added to the MATS directory after GUI initialization will not appear in the GUI until either the GUI is restarted or its module data is refreshed via **Configuration** → **Refresh Module Data** in the window menu.

## 2.2.2 Additional Requirements for Data Readers

### Main Function Parameters

A data reader's main function must take in two inputs. The first is a file name structure, which is described in detail in Table 2.7. The second is a ground truth file path string, which will be empty if no ground truth has been provided. The sole output is a MATS input structure as defined in the tables in Section 1.2.1.

### Description File Details

The data reader's description file is more complicated than other module types in that the fields defined in the file are linked to a greater degree. Where other modules' fields enable options that are largely independent, most of the data reader's fields must be compatible with each other or else errors will occur.

[1]Each of these fields should contain the same number of elements, and consistent order should be maintained among them. For example, the first filter listed in **band_regexps** should correspond to the first field listed in **field_names**. Elements should be delimited by semicolons, even if there is only one element in each of these fields.

[2]If both port and starboard data are contained in the same file, use the same expression for both fields. One of these fields should be empty if it is known that each file in a group of data is from the same side.

[3]Either 1/0 or 'yes'/'no' will be properly interpreted by the GUI

The purpose of the data reader's description file is to inform the creation of the *file name structure vector*, the comprehensive list of all files that will be processed during the course of a particular ATR run. Consequently, most of the fields in the description file specify what kinds of data are required and how that data is stored. For example, `field_names` defines which data types (e.g., different sensors and/or frequency bands) are expected for this file format. For each type defined in `field_names`, the corresponding regular expression in the `band_regexps` field will be used to create a list of data files that contain that data type. Care should be taken when crafting these expressions or else errors may result from attempt to load invalid files.

The `band_replace_strs` field is used to group these data files into logical matching sets; by replacing the string detected using `band_regexps` with the items in `band_replace_strs`, the corresponding files for other data types can be derived. The result of this process is a set of complete file name structures, meaning that none of them are missing any of the specified data types. Because all three of these fields refer to same set of data types, they must all have the same number of entries.

In addition, port and starboard data files are identified using the regular expressions in the fields `port_regexp` and `stbs_regexp`, respectively. If the same file contains both both port and starboard data, these expressions should be identical. If it is known that all of the data files belong to one side, then the other's expression can be blank.

### 2.2.3 Additional Requirements for Detectors

#### Main Function Parameters

A detector's main function, `det_[MODULE_SUBDIR].m`, must take in two inputs (an input structure and a detector configuration structure) and return two outputs (a contact list and an extra contact data structure) as defined in Section 1.2 (Data Structures). The new detector should only operate on familiar sensor types; if an unrecognized sensor is chosen, it should produce an error to prevent producing potentially misleading results. The easiest way to do this is to use the `for_sensors` entry in the detector's description file to list valid sensor types.

A detector should also initialize all of the fields in the contact list to default values before returning them. Most of the fields can be initialized automatically using existing subroutines. An example of this is shown in the sample detector file `Detectors\Test\det_Test.m`.

#### Detector Configuration Files

At least one *detector configuration file* must be created before the detector can be run successfully. Each of these files contains detector parameters that may change over different sensors and environments. The data is stored in the fields of a structure, as summarized in Table 2.5. All configuration files must be saved in a `.mat` format with a `.dconfig` file extension. The configuration that is chosen from the GUI's 'Det.

Table 2.5: Fields of a Detector Configuration File

| **Field Name** | **Description** |
|---|---|
| `.ver` | version number |
| `.note` | string describing configuration |
| `.time` | time stamp (from MATLAB®'s `now` function) |
| `.det` | string containing detector name |
| `.sensor` | sensor string (should match one of the strings in the GUI's drop-down menu) |
| `.params` | structure for other detector parameters |

configuration' selection menu will be used at run time. *(Refer to Section 1.3.3 for more information about the GUI).*

### Image Normalization

A subroutine for image normalization (`ATR_Core\normalize_images.m`) is included to assist in module development. If this function is called, the `.normalizer` field of the contact list should be set to `'NSWC'`. If a different function is used, a short string identifier should be used instead. If no image normalization is performed, use an empty string (i.e., `''`).

## 2.2.4 Additional Requirements for Features

While earlier diagrams have depicted the feature extraction module as a single entity for clarity, this processing stage could in fact be several feature extraction modules executed in succession, as shown in Figure 2.2. This allows for more flexibility in algorithm selection and enables the feature modules to be simpler and more compact. Thus, it may be advantageous to break a feature module into several smaller ones so that any or all of the feature subsets can be used as desired.

The feature modules are executed largely independently of each other. However, it is important to note that the contact list is passed from one module to the next in order to accumulate feature data. Therefore, *feature data should be appended to their respective fields in the contact list.*

### Main Function Parameters

A feature module's main function, `feat_[MODULE_SUBDIR].m`, takes in three or four input arguments (contacts, extra contact data, optional field data, and modules configuration) and return one output (the contact list with added features) as defined in Section 1.2 (Data Structures). The fourth input argument is optional depending on whether the feature module supports optimization (see Section 2.2.4).

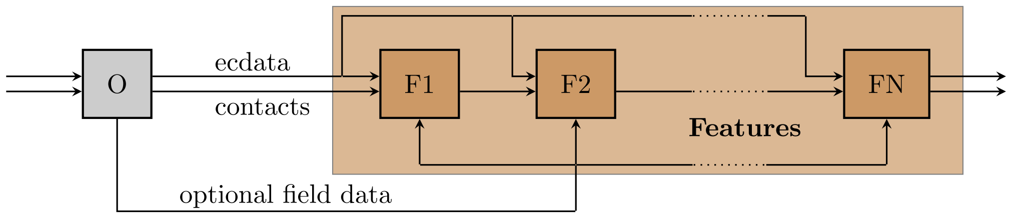
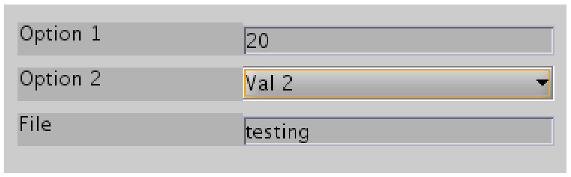


Figure 2.2: Detailed View of Feature Extraction Process

(a)

(b)

```
cfg.opt1 = {'Textbox', 'Option 1', 20};
cfg.opt2 = {'Drop', 'Option 2', ...
            'Val 1|Val 2|Val 3', 2};
cfg.fn = {'Textbox', 'File', 'testing'};
```



| | |
|---|---|
| Option 1 | 20 |
| Option 2 | Val 2 |
| File | testing |

Figure 2.3: (a) An example parameter description structure. (b) UI created from parameter description structure shown in (a).

### Feature Space Optimization Function

As discussed in Section 1.3.9 (Classifier Training), some feature extraction methods are specific to a particular system, or have parameters that can be tuned to alter performance. A feature extraction optimization function can be defined to allow an end user to train and alter specific feature module parameters from the Classifier Training GUI. This optimization function behaves like a feature set training function, except it is only called by the Classifier Training GUI for the purpose of classifier training. Hence, it is only called with contact information that is fully labeled. The function should be placed in the module's root directory with the feature extraction function and named `opt_[MODULE_SUBDIR].m`.

The feature extraction optimization function takes the same inputs as the main feature extraction function with the addition of an optional fourth argument, which is a feature parameter structure that is created by the Classifier Training GUI. This function has a very specific behavior based on the number of arguments that are passed. If it is called without input arguments, then it should return one output argument: a parameter description structure, which is described below. If it is called with all four parameters, then the function should train on the input contact information (which is given in the first three input arguments) and return two outputs. The first is the contact list with features added, similar to the output of the feature extraction function. The second is a feature configuration structure, which is written into the classifier training file and is later passed as a fourth argument to the feature extraction function during ATR processing. This saves the trained feature module's configuration for future feature extraction.

The classifier training GUI uses the feature parameter structure generated by the feature extraction optimization function to construct an input GUI for a feature module's parameters. Every field in the structure is considered a parameter. The value of the field is a 1x4 cell array containing a description of the User Interface (UI) object to be created for it:

- The first element of the cell array determines the type of the UI object, either `Drop` or `Textbox`. A `Drop` is a UI drop down menu containing a specified number of selectable values. A `Textbox` is a UI input box in which the user can specify a string value. If a numerical value is expected, the optimization function must convert the string to a numerical value.
- The second value of the cell array is a string that is used as the title for the UI object.
- The third cell value is used to populate the UI object. If the UI object is a `Drop`, then this value must be a string containing selectable values separated by the `'|'` character. If the UI object is a `Textbox`, then its default value is set as this value.
- The fourth cell value is used by the `Drop` UI object to select a default value from the list given in the third cell value.

When the classifier training begins, any changes to the default UI object's value are put into a feature configuration structure. This structure contains **only** the fields from the parameter description structure that were changed by the user and those fields' corresponding values. This structure is passed to the feature optimization function as the fourth argument, thereby passing the users desired parameter setting to the feature optimization function. An example of a parameter description structure and the UI object generated from it are shown in Figure 2.3. A prototype feature optimization function is presented in Figure 2.4 to illustrate its structure.

### 2.2.5 Additional Requirements for Classifiers

#### Main Function Parameters

The main function of a new classifier, `cls_[MODULE_SUBDIR].m`, should take in two inputs (a contact list and a classifier configuration file name) and return one output (the contact list with updated classification data). Since classifiers can be trained on different feature sets, additional steps are required to generate the configuration file.

#### Classifier Configuration Files

At least one *classifier configuration file* must be created before the classifier can be run successfully. Each of these files contains data about the features that are used in the configuration. The data is stored in the fields of a structure, as summarized in Table 2.6. All configuration files must be saved in a `.mat` format with a `.cconfig` file extension. In addition, the default data file should be indicated with `default` in the

```
function [out, cfg] = opt_[SUBDIR](con, ecdata, ofdata, cfg)
    options.opt1 = {‘Textbox’, ‘Option 1’, 20};
    options.opt2 = {‘Drop’, ‘Option 2’, ‘Val 1|Val 2|Val 3’, 2};
    options.fn = {‘Textbox’, ‘File’, ‘testing’};

    % If we are called with no arguments we return a structure contains
    % configurable options, which can be used to for the cfg input.
    if nargin==0
        out = options;
        cfg = [];
        return;
    end

    % The remainder of the function should be devoted to training the feature
    % module and computing the features. Any additional configurations done
    % during training can be added to the ‘cfg’ structure, which will be saved
    % in the cconfig file generated from the classifier train and later passed
    % to ‘feat_[SUBDIR]’ as a fourth input argument when MATS is run.

    % The featConfigure function is a MATS function for selecting the default
    % options from a feature parameters structure (‘options’) and applying a
    % user configuration (input argument ‘cfg’). All user configurations a
    % given as strings. So they must be converted it they should be anything
    % other than a string.
    cfg = featConfigure(options, cfg);
    if ischar(cfg.opt1)
        cfg.opt1 = str2num(cfg.opt1);
    end

    % ‘out’ now becomes our contact list with the features computed.
    out = con;
end
```

Figure 2.4: Prototype of Feature Extraction Optimization Function

Table 2.6: Fields of a Classifier Configuration File

| **Field Name** | **Description** |
|---|---|
| `.ver` | version number |
| `.note` | string describing configuration |
| `.time` | time stamp (from MATLAB®'s `now` function) |
| `.detector` | string for which detector can be used with this file (use detector name if using detector features; otherwise use 'all') |
| `.featsets` | N x 1 cell array of strings indicating which feature modules to run (each string should match a valid feature module's subdirectory) |
| `.featver` | 1 x N array of each feature module's version numbers |
| `.featcfg` | N x 1 cell array containing each feature modules optional parameters (if any) |
| `.featNorm` | 2 x `sum(lens)` array of feature normalization weighting parameters |
| `.clscfg` | classifier configuration parameter structure (will be passed into classifier at runtime) |
| `.lens` | 1 x N array containing lengths of each feature module's feature vector |
| `.sensor` | sensor string (should match one of the strings in the GUI's drop-down menu) |
| `.cls` | string containing classifier name |

file name somewhere. The configuration that is chosen from the GUI's 'Class. configuration' selection menu will be used at run time. *(Refer to Section 1.3.3 for more information about the GUI).*

### Training Functions

A classifier should also have a training function to assist in the creation of classifier configuration files. The training function should have a function definition of the form `trn_[MODULE_SUBDIR](labels, features, data_fname)`, where `labels` is a $1 \times N$ row vector containing ground truth information for $N$ contacts, `features` is an $L \times N$ matrix containing the feature vectors of these contacts as columns, and `data_fname` is the name that the new classifier data file will have. `labels` and `features` will be extracted from prior results (i.e., the `IO_*.mat` files from a prior run).

### 2.2.6 Feedback

Additional steps must be taken to incorporate operator feedback (i.e. data from the user/operator intended to improve performance) into classifier modules. Feedback can be acquired either in series (by using either the basic feedback GUI to acquire feedback, or ground truth information to simulate it), or in parallel (by using some other external feedback GUI). The series case is much simpler because it avoids conflicts between the ATR and the feedback acquisition process, as only one of the two processes can modify the contact list at a time. However, in that scenario, both processes must wait for the other to finish, resulting in a longer overall runtime. Thus, the parallel approach is often preferred despite its increased complexity.

In either case, the classifier can query the operator via the `.opfeedback.opdisplay` fields of the contacts in the contact list. By setting these to a positive value the classifier can indicate that the operator should confirm or reject a given target. In response, the operator rates each contact in turn on a five-point scale, with a value of five corresponding to a confident target classification; three, an ambiguous case; and one, a confident non-target classification. These ratings are stored in `opfeedback.opconf`.

Once the operator has rated a contact, this information needs to be relayed back to the ATR where it can inform the classifier. The operator decisions are written to a dedicated *feedback file*, the contents of which will be read during the next iteration of the ATR processing. The information contained in the feedback file will be incorporated into the contact list, and then the feedback file will be cleared of any data. This process is covered in more detail in Appendix 2.3 (ATR and Operator Modules).

A classifier should never attempt to reclassify a contact that has been viewed by the operator. After the operator has evaluated all of the contacts derived from one image, it is assumed that he has made any necessary corrections to erroneous calls. The last contact that the operator has viewed can be easily determined by finding the latest negative value for `.opdisplay`.

## 2.3 ATR and Operator Modules

In order to successfully integrate a feedback classifier, some additional details of the MATS architecture must be understood. Otherwise, unexpected behavior may result, and the causes may be difficult to identify. This section assumes that the parallel mode of feedback acquisition is desired to avoid unnecessary waiting.

The mechanism by which the feedback is acquired from the operator is called the *operator process* and runs in parallel with the main ATR process. Because these processes run independently of each other, they must communicate with each other via common files. The operator process can check periodically for new output files (i.e., new `IO_*.mat` files located in the output directory) to get information about new contacts that have been recently created by the ATR process. After obtaining the operator's feedback on the new contacts, the operator process should record the feedback in order to pass it on to the classifier. This can be done using existing subroutines that edit the saved contacts in the contact buffer file directly.

The contact buffer file contains contacts that the operator has still not yet seen. At the end of the ATR function, all contacts that have recently been evaluated by the operator (as indicated by changes in their `.opfeedback.opconf` values) are removed from the contact buffer file, and all contacts from the recently-completed image are added to it.

Internally, two indices that determine how a contact's data may be modified are monitored. The *last locked index* is the index of the last contact deemed to have completed the feedback process. This index can be found by locating the last contact in the front portion of the contact list with a non-negative `.opfeedback.opconf` value. Contacts before this index have received feedback since the last invocation of the ATR, and will be removed from the contact buffer at the end of this iteration.

The *last viewed index* is the index of the last contact that the operator has had the opportunity to see. What this means depends on how the feedback is acquired. If the contacts are presented sequentially, then the last viewed index corresponds to the index of the last contact presented. If the operator sees all of the contacts from one image simultaneously, then this would be the latest index of the contacts on the screen, regardless if the operator has really examined or commented on that contact. The contact at the last viewed index is the last contact in the front portion of the list with a negative value of `.opfeedback.opdisplay`. (Alternatively, it is the contact before the earliest one with a non-negative value of `.opfeedback.opdisplay`.) When the ATR runs, any contact that has not been viewed will be passed into the classifier for classification.

Figure 2.5 illustrates these concepts for a sample contact list. In this scenario, all contacts from the same image are viewed simultaneously. Each column in the array represents a contact, and each row represents an operator feedback field (i.e., `.opdisplay` or `.opconf`). The contents of each cell indicates the sign of the feedback field listed on its row for that contact. In this example, `contacts(2).opfeedback.opconf` is zero, and `contacts(2).opfeedback.opdisplay` is negative. The locked contacts are shaded red; the unlocked contacts are shaded either white or blue. The blue contacts represent contacts from future images that have already been processed by the ATR, but have not been viewed by the operator, and are hence able to be reclassified. The white contacts are currently on the operator's screen. Note that if the contacts were being evaluated sequentially, the white region would consist of at most one contact. The last locked index and last viewed index are the indices of the last contact in the red and white regions, respectively. By modifying the operator feedback fields, these indices advance and the contacts' data becomes more protected.

## 2.3.1 Modifying the Contact List with Subroutines

Several functions located in the `ATR␣Core` subdirectory of the MATS library can help in communicating the operator feedback to the ATR process. The general algorithm for this is:

1. Read value of `contacts(k).opfeedback.opdisplay`. If this value is positive, then feedback is requested for this contact. *(Note: Refer to Table 1.7 for more information.)*

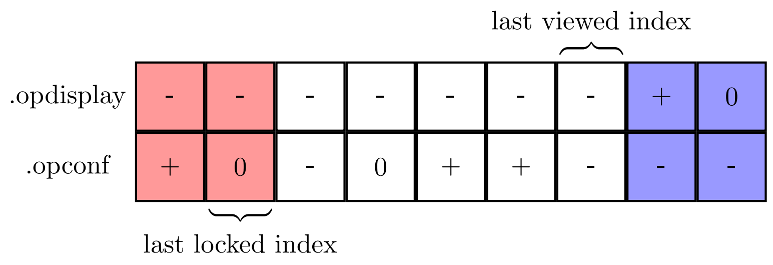


Figure 2.5: MATS Operator Feedback Scheme.

2. Query the operator to acquire feedback (i.e., `contacts(k).opfeedback.opconf`). MATS currently supports a five-point scale, with a five likely being a target and a one likely being clutter.

3. Wait while the file `bkup_atr_busy.flag` exists using the `wait_if_flag` function. When `bkup_atr_busy.flag` is present, the ATR process is busy modifying the locked and unlocked files.

4. Create `bkup_gui_busy.flag` with the `write_flag` function. Similarly, the presence of `bkup_gui_busy.flag` informs the ATR process that the operator is about to change the contents of the locked file and/or the unlocked file. The ATR process will then wait until the contact list files are again available before trying to access them.

5. Check if any new contacts are available by searching for new I/O files in the output directory. A new file in the directory indicates that the contact list may have been changed.

6. If new contacts are available, load the latest version of the contact list using the `read_backup` function.

7. Edit the contact list to incorporate operator confidence (`.opconf`).

8. Save contact list via the `write_backups` function.

9. Delete `bkup_gui_busy.flag` with `delete_flag`. Failure to delete `bkup_gui_busy.flag` will cause the ATR to hang indefinitely!

Notes:

- When a contact is viewed, 10 should be subtracted from its `.opfeedback.opdisplay` value, and the saved lists should be updated using Steps 3-9. This will prevent contacts that are currently being viewed from being reclassified.

- Details of the functions mentioned above can be found in Appendix 2.5.

- The ATR process contact buffer file is located at `[MATS_ROOT]\bkupedit.txt`. Use this file when calling `read_backups` or `write_backups`.

## 2.4 Supporting Data Structures

This section highlights some important data structures used by MATS. Knowledge of these structures should not be necessary for normal MATS operation, but it might be needed to make modification or to troubleshoot.

### 2.4.1 File Name Structure Vector

The *file name structure vector (FNSV)* contains all of the information about what files are included in an ATR run. The vector as a whole represents the entire data set to be processed. Each individual element of the FNSV corresponds to a particular set of one or more images that span the same spatial area. ATR is performed on each of these sets in turn until all of them have been completed.

The fields contained in the FNSV are listed in Table 2.7. Note that some of the field names depend on which frequency bands are associated with that type data. These are defined in the appropriate data reader's description file (Table 2.4).

Table 2.7: Fields of a File Name Structure

| **Field Name** | **Description** |
|---|---|
| `.[band]` | name of file containing data of a certain band |
| `.[band]_path` | directory containing file specified in `.[band]` |
| `.side` | side of vehicle = {'PORT', 'STBD'} |
| `.label` | string by which this image will be identified |

### 2.4.2 Module Data Structure

The *module data structure* is a repository of all information known about all modules within the MATS directory hierarchy. This structure is created during the GUI initialization process after exhaustively searching each of the module pool directories (e.g., `\Detectors`) for usable modules. Table 2.8 provides an overview of its contents.

The GUI then uses this data, along with other logic, to determine which modules should be listed in the appropriate module selection menus. For example, classifiers lacking a valid configuration file (i.e., they have not been trained) for a given sensor will automatically be pruned from the list of classifiers in the main GUI because they cannot be run as is; however, they will remain in the list in the classifier training GUI.

Table 2.8: Contents of Module Data Structure

| .read | .det | .feat | .cls | .perf | .cor | .seg | .est | Module Data Subfield |
|---|---|---|---|---|---|---|---|---|
| X | X | X | X | X | X | X | X | `name`: string identifier for module (equivalent to module's subdirectory name) |
| X | X | X | X | X | X | X | X | `func`: function handle of algorithm's main function |
| X | X | X | X |  |  | X |  | `info`: structure containing data read from description file (about.txt) (See Table 2.4) |
|  | X |  | X |  |  | X |  | `config_fn`: cell array of filenames of local configuration files for this module |
|  | X |  | X |  |  | X |  | `config`: cell array of data contained in the files named by `config_fn` |

## 2.5 Notable Functions

#### MATS_GUI

`function MATS_GUI()` – Main function for the MATS Configuration GUI. No inputs or outputs.

### 2.5.1 Configuration GUI Subroutines

#### import_module_data

`function mod_data = import_module_data(mats_root, varargin)` – Scans MATS directory structure for all available algorithm modules and creates a module data structure that serves as an index of valid modules, how they are configured, and what they require.

- `mats_root` = root-level MATS directory
- `varargin` (optional) = if `varargin{1}` is `'noconfigok'`, modules without a valid configuration file are permitted; otherwise, they will be purged from the list
- `mod_data` = module data structure

#### gen_config_list

`function list = gen_config_list(mod_type, mod_param)` - Returns a list of all configuration files for a given module and the current configuration.

- `mod_type` = module type:

  - '`sconfig`' for image segmenter configuration files
  - '`dconfig`' for detector configuration files
  - '`cconfig`' for classifier configuration files

- `mod_param` = either a GUI handle from which a module data structure can be extracted, or a module data structure itself

#### gen_module_list

`function list = gen_module_list(mod_type, mod_param)` - Returns a list of all modules of a given type that are compatible with the current configuration.

- `mod_type` = module type:
  - '`read`' for data reader modules
  - '`seg`' for image segmentation modules
  - '`est`' for parameter estimation modules
  - '`read`' for data reader modules
  - '`det`' for detector modules
  - '`feat`' for feature modules
  - '`cls`' for classifier modules
  - '`perf`' for performance estimation modules
  - '`cor`' for contact correlation modules

- `mod_param` = either a GUI handle from which a module data structure can be extracted, or a module data structure itself

### 2.5.2 Detector Subroutines

#### initialize_contact_data_lists

`function [contacts, ecd_vec] = initialize_contact_data_lists(N)` - Initializes a contact list and ecdata list with default values and matching uuids.

- `N` = desired length of data lists
- `contacts` = contact list (See Table 1.6)
- `ecd_vec` = list of extra contact data structures (See Table 1.8)

#### detsub_contact_fill

`function [contacts, ecd_vec] = detsub_contact_fill(contacts, ecd_vec, k, input_struct)` – Fills in many contact fields of `contacts(k)` with default values or values coming directly from the input structure.

- `contacts` = contact list (See Table 1.6)
- `ecd_vec` = list of extra contact data structures (previously included in contact list) (See Table 1.8)
- `k` = index in contact list
- `input_struct` = input structure (See Table 1.1)

#### detsub_gt

`function contacts = detsub_gt(contacts, input_struct)` – Incorporates the ground truth data from an input structure into the contact list. (Note: This function assumes that all of the contacts in `contacts` are in the image represented by `input_struct`.)

- `contacts` = contact list (See Table 1.6)
- `input_struct` = input structure (See Table 1.1)

### 2.5.3 External Feedback Reference

#### wait_if_flag

`function wait_if_flag(flagname)` – Causes execution to wait while the file(s) defined in `flagname` exist.

- `flagname` = either a string representing the filename of the flag to wait for, or a cell array containing multiple such strings

#### write_flag

`function write_flag(filename, msg_on)` – Writes a tiny, empty file whose sole purpose is merely to exist. This is used to inform processes of which files are currently being used.

- `filename` = the name of the file to be created
- `msg_on` = 0 or 1, enables debugging messages

#### delete_flag

`function delete_flag(filename, msg_on)` – Deletes a flag file.

- `filename` = the name of the file to be deleted
- `msg_on` = 0 or 1, enables debugging messages

### read_backups

`function c_list = read_backups(fn_l, fn_u, msg_on)` – Reads the backup files specified in `fn_l` and `fn_u` and imports the data into a contact list.

- `fn_l` = file name of the backup file used to store unchangeable (locked) contacts
- `fn_u` = file name of the backup file used to store contacts that can still be changed (unlocked)
- `msg_on` = 0 or 1, enables debugging messages
- `c_list` = a contact list containing all of the data contained in the input

### write_backups

`function write_backups(c_list, lock_ind_old, lock_ind_new, fn_l, fn_u, msg_on)` – Writes the contact list (`c_list`) over two files: `fn_l`, for past contacts that cannot be changed again, and `fn_u`, for more recent contacts that may be reclassified again.

Normally, the locked list is only appended to since, by definition, the contacts within the locked list cannot change. However, it may be desirable to manually remove a contact that causes problems for the ATR, which would also require removing it from the locked backup. In this case, call this function with `lock_ind_old == 0`. This will mimic the first call to this function and overwrite the locked backup with all contacts with indices up to and including `lock_ind_new`.

- `c_list` = contact list (See Table 1.6)
- `lock_ind_old` = index of last locked contact from previous call to `write_backups`
- `lock_ind_new` = index of last locked contact from current call to `write_backups`
- `fn_l` = file name of the backup file used to store locked contacts
- `fn_u` = file name of the backup file used to store contacts that can still be updated
- `msg_on` = 0 or 1, enables debugging messages.

### find_last_locked

`function lastlock_index = find_last_locked(contacts, show_details)` – Determines the last contact to evaluated by the operator by scanning the `.opconf` values.

- `contacts` = contact list (See Table 1.6)
- `show_details` = 0 or 1, enables debugging messages
- `lastlock_index` = index of the last contact to be evaluated by the operator

**find_last_viewed**

`function lastview_index = find_last_viewed(contacts, new_img_ind, show_details)` – Determines the last contact to be seen by the operator by scanning the `.opdisplay` values.

- `contacts` = contact list (See Table 1.6)
- `new_img_ind` = first index of current image; search will start at the contact before this location
- `show_details` = 0 or 1, enables debugging messages
- `lastview_index` = index of the last contact that was seen by the operator

## 2.6 Data Files

This section contains an overview of the data files that are involved in the execution of MATS. These files and their interactions are summarized in Figure 2.6. The blocks in the diagram represent major MATS functions that are relevant to reading and writing data files, and the circles correspond to the files themselves. The solid lines indicate a direct flow of data, and the dashed lines show other dependencies.

Each ATR module has a *description file* (`about.txt`) that informs the GUI of the module's requirements, as specified in Table 2.4 in Section 2.2. For example, a classifier's description file indicates whether it can benefit from user feedback; if it can, then the GUI must run certain subroutines to facilitate communication with the user. The description files are indicated by (A)'s in Figure 2.6. Each file is of the same color as the module it describes.

The parameters from the description files are imported into the GUI's module data structure during GUI initialization. When the user chooses a particular module (or module configuration file) via the GUI, other configuration parameters are adjusted to accommodate the requirements listed in the description files. For example, when a classifier configuration file is selected, the features used to create that classifier configuration are automatically selected as well.

After the user initiates ATR via the GUI's start button, the configuration parameters are passed to the processing loop. The loop loads the first input data file (not shown), converts the data into a MATS input structure, and then begins the ATR processing using the set of modules and module configuration files that were selected in the GUI. The input structure flows through the optional environmental modules to the detector module (D), which yields two outputs, the *contact list* and the *extra contact data*. The contact list contains all the essential variables (e.g., position, classification, potential ground truth), while the extra contact data contains the remaining standard data fields that are associated with contacts. Both of these data structures are used by one or more feature modules (F) and any optional modules (O) that the feature modules may require.

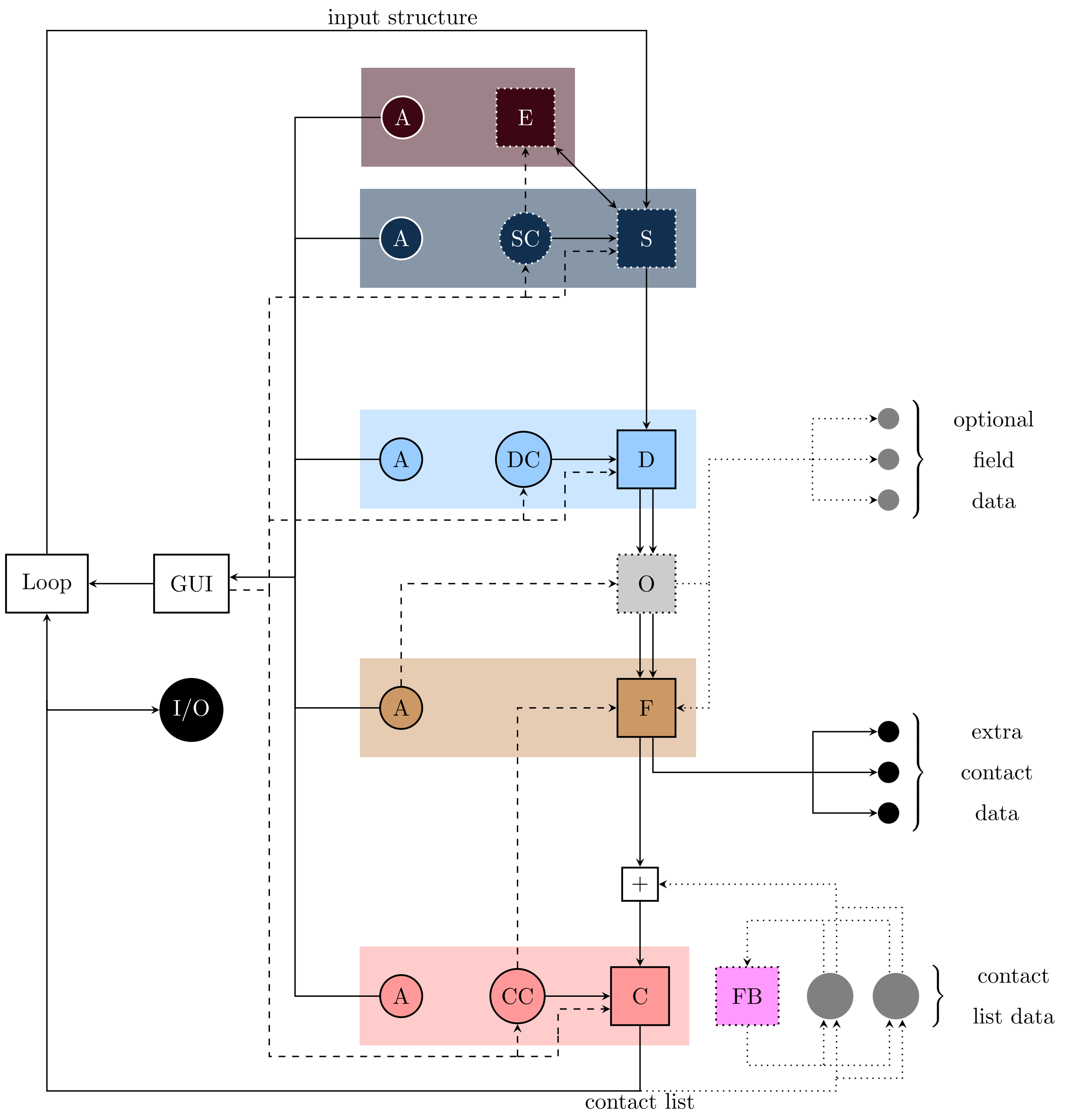


Figure 2.6: MATS Data Files

Table 2.9: Required Ground Truth File Column Headings

| **Binary** | **Multiclass** | **GT Column Heading** |
|---|---|---|
| X | X | `X`: x-coordinate of object (in pixels). Negative values indicate object is on port side. |
| X | X | `Y`: y-coordinate of object (in pixels). |
| X | X | `FILENAME`: name of the source image data file. |
|  | X | `TYPE`: object type. |
|  |  | `OBJNUM`: object number. Values should be identical for all instances of a given object. |

At this point, the paths of the data structures diverge. The extra contact data is saved to disk, with each file containing the data for an individual contact. Similarly, the optional field data may also be saved in this fashion, depending on the user's configuration. The contact list, which contains the contacts' features, continues on to the classifier (C). The classifier uses the *classifier configuration file* (CC), which is created during the training process, to determine the classification of each contact in the list. If a feedback classifier is being used, the contact data is saved over two files, one for contacts that the user has not yet seen and are thus reclassifiable, and another for those for which the user has already provided feedback. At this point, ATR processing is complete.

Finally, the processing loop function saves a copy of the contact list (and possibly other data as well) in an I/O file (`IO*.mat`). Once all of the input data files have been processed, the folder of resulting I/O files and the corresponding folder of extra contact data can be analyzed using several built-in tools, such as the ROC curve generator.

### 2.6.1 Ground Truth Files

Ground truth files contain lists of image locations of objects that should be identified by the ATR. A ground truth file is not necessary for basic ATR operation, but it is required to train algorithms or to evaluate performance.

MATS supports a flexible ground truth format. The formatting of the ground truth data does not matter so long as all of the necessary information is present. There must be a comment line (starting with %) at the beginning of the file that defines what the columns of data are. The required entries for this comment are listed in Table 2.9, and a sample heading is shown as an example in Figure 2.7 for reference. Note that the third entry will be ignored because it has been commented out. In addition, a comment line of '`% MULTICLASS`' will indicate that this file is intended to have multiple object classes in it.

```
% COLUMNS: X Y FILENAME
1230 456 Image0001.mat
234 567 Image0001.mat
% 543 432 Image0002.mat
678 987 Image0003.mat
```

Figure 2.7: Sample Ground Truth File

# Appendix A

# Version History

## A.1 Version 3.2

### A.1.1 Significant Changes

**Graphics:**

- Modified image display to allow showing imagery for multiple sensors
- Improved GUI positioning on smaller monitors

**General Processing:**

- Added environmental modules
- Removed locked contacts from contact backup file during feedback mode

**Analysis Tools:**

- Modified results analysis to account for sweetspot region

**Miscellaneous:**

- Documentation update

## A.2 Version 3.1

### A.2.1 Significant Changes

**Graphics:**

- Modified graphics code to be compatible with MATLAB's graphics overhaul in R2014b

**General Processing:**

- Added `.uuid` field to contacts and ecdata structures
- Modified classifier configuration files: Data that was previous stored in the auxiliary .mat file (defined by the field `.cdfn`) is now embedded in the .cconfig file as the substructure `.clscfg`

**Analysis Tools:**

- Added ROC curve option for combining multiple sets into one

**Miscellaneous:**

- Documentation update

## A.3 Version 3.0

### A.3.1 Significant Changes

**Graphics:**

- New configuration GUI - more compact, extensible, and robust; increased integrity checking of modules
- Added sweet spot markers to image plots - shows ATR operating region

**General Processing:**

- Chunking restructuring - chunking wrapped around only detector and performance estimation modules instead of the entire ATR stream
- Detector training - allows for variable configuration of detector modules based on different sensors, environments, etc.
- Data integrity module - will skip ATR if vehicle is turning (Edgetech only) or if height anomalies are detected

**Distributed Processing:**

- Added logging capability - prompt messages are now duplicate in log file
- Added failure option - files can now fail gracefully and be reset via progress GUI
- Improved color scheme for task progress GUI - see Figure 1.17 for new colors

**Analysis Tools:**

- Modified existing tools to route outputs to a new performance summary file `PERF_STATS.mat`, which is located in ATR tasks' output directories
- Added routine to generate a LaTeX .pdf summarizing performance metrics
- Added aggregate detection rate and aggregate false alarms metrics
- Merged single- and multi-ROC functions into a single tool

**Miscellaneous:**

- File organization - many files in `ATR Core` and `GUI` directories have been grouped into subdirectories
- Ground truth files may now have comment lines in data
- db_tool: Contact list filtering function `filter_db` now handles compound expressions
- Documentation update

DISTRIBUTION

| | |
|---|---|
| Defense Technical Information Center<br>ATTN:  DTIC-0<br>8725 John J. Kingman Road<br>Fort Belvoir, VA 22060-6218 | 1 |
| Office of Naval Research<br>ATTN:  Dr. Jason Stack<br>Mine Warfare & Ocean Engineering Programs Code 32, Suite 1092<br>875 North Randolph Street<br>Arlington, VA 22203-1995 | 1 |
| Naval Surface Warfare Center, Panama City Division<br>ATTN: Technical Library<br>110 Vernon Avenue<br>Panama City, FL 32407-7001 | 2 |
| Naval Surface Warfare Center, Panama City Division<br>ATTN: Code X23, Mr. Derek R. Kolacinski<br>110 Vernon Avenue<br>Panama City, FL 32407-7001 | 1 |
| Naval Surface Warfare Center, Panama City Division<br>ATTN: Code X24, Mr. Bradley C. Marchand<br>110 Vernon Avenue<br>Panama City, FL 32407-7001 | 1 |

NAVSEA
WARFARE CENTERS
PANAMA CITY